Livro de receitas do GNU Guix

Tutoriais e exemplos para usar o Gerenciador de Pacotes Funcional do GNU Guix

Desenvolvedores do GNU Guix

Sumário

1 Tutoriais sobre Scheme

GNU Guix foi escrito na linguagem de programação de uso geral Scheme, e muitos de seus recursos podem ser acessados e manipulados programaticamente. Você pode usar Scheme para gerar definições de pacotes, modificá-los, construí-los, implantar sistemas operacionais inteiros, etc.

Conhecer o básico de como programar com Scheme irá desbloquear muitos dos recursos avançados que o Guix oferece — e você nem precisa ser um programador experiente para usá-los!

Vamos começar!

1.1 Um curso intensivo de Scheme

Guix usa a implementação Guile do Scheme. Para começar a brincar com a linguagem, instale-a com `guix install guile` e inicie um *REPL*—abreviação de *read-eval-print loop* (`https://pt.wikipedia.org/wiki/REPL`)—executando `guile` na linha de comando.

Alternativamente, você também pode executar `guix shell guile -- guile` se preferir não ter o Guile instalado em seu perfil de usuário.

Nos exemplos a seguir, as linhas mostram o que você digitaria no REPL; linhas que começam com "⇒" mostram resultados de avaliação, enquanto linhas que começam com "⊣" mostram coisas que são exibidas. Veja Seção "Using Guile Interactively" em *Manual de referência do GNU Guile*, para mais detalhes sobre o REPL.

- A sintaxe de Scheme se resume a uma árvore de expressões (ou *s-expression* no jargão Lisp). Uma expressão pode ser um literal, como números e strings, ou um composto que é uma lista entre parênteses de compostos e literais. `#true` e `#false` (abreviados `#t` e `#f`) representam os booleanos "true" e "false", respectivamente.

  Exemplos de expressões válidas:

  ```
  "Hello World!"
  ⇒ "Hello World!"

  17
  ⇒ 17

  (display (string-append "Olá " "Guix" "\n"))
  ⊣ Olá Guix!
  ⇒ #<unspecified>
  ```

- Este último exemplo é uma chamada de função aninhada em outra chamada de função. Quando uma expressão entre parênteses é avaliada, o primeiro termo é a função e o restante são os argumentos passados para a função. Cada função retorna a última expressão avaliada como valor de retorno.
- Funções anônimas —*procedures* na linguagem do Scheme — são declaradas com o termo `lambda`:

  ```
  (lambda (x) (* x x))
  ⇒ #<procedimento 120e348 em <porta desconhecida>:24:0 (x)>
  ```

O procedimento acima retorna o quadrado do seu argumento. Como tudo é uma expressão, a expressão `lambda` retorna um procedimento anônimo, que por sua vez pode ser aplicado a um argumento:

```
((lambda (x) (* x x)) 3)
⇒ 9
```

Os procedimentos são valores regulares, assim como números, strings, booleanos e assim por diante.

- Qualquer coisa pode receber um nome global com `define`:

```
(define a 3)
(define square (lambda (x) (* x x)))
(square a)
⇒ 9
```

- Os procedimentos podem ser definidos de forma mais concisa com a seguinte sintaxe:

```
(define (square x) (* x x))
```

- Uma estrutura de lista pode ser criada com o procedimento `list`:

```
(list 2 a 5 7)
⇒ (2 3 5 7)
```

- Os procedimentos padrão são fornecidos pelo módulo `(srfi srfi-1)` para criar e processar listas (veja Seção "SRFI-1" em *Manual de referência do GNU Guile*). Aqui estão alguns dos mais úteis em ação:

```
(use-modules (srfi srfi-1)) ;importar procedimentos de processamento de lista

(append (list 1 2) (list 3 4))
⇒ (1 2 3 4)

(map (lambda (x) (* x x)) (list 1 2 3 4))
⇒ (1 4 9 16)

(delete 3 (list 1 2 3 4))        ⇒ (1 2 4)
(filter odd? (list 1 2 3 4))     ⇒ (1 3)
(remove even? (list 1 2 3 4))    ⇒ (1 3)
(find number? (list "a" 42 "b")) ⇒ 42
```

Observe como o primeiro argumento para `map`, `filter`, `remove` e `find` é um procedimento!

- O *quote* desativa a avaliação de uma expressão entre parênteses, também chamada de expressão S ou "s-exp": o primeiro termo não é chamado sobre os outros termos (veja Seção "Expression Syntax" em *Manual de referência do GNU Guile*). Assim, ele efetivamente retorna uma lista de termos.

```
'(display (string-append "Hello " "Guix" "\n"))
⇒ (display (string-append "Hello " "Guix" "\n"))

'(2 a 5 7)
⇒ (2 a 5 7)
```

- O `quasiquote` (`` ` ``, uma crase) desativa a avaliação de uma expressão entre parênteses até que `unquote` (`,`, uma vírgula) a reative. Assim, nos fornece um controle refinado sobre o que é avaliado e o que não é.

```
`(2 a 5 7 (2 ,a 5 ,(+ a 4)))
⇒ (2 a 5 7 (2 3 5 7))
```

  Observe que o resultado acima é uma lista de elementos mistos: números, símbolos (aqui `a`) e o último elemento é uma lista em si.

- Guix define uma variante de expressões simbólicas (S-expressions) com esteróides chamada *G-expressions* ou "gexps", que vem com uma variante de `quasiquote` e `unquote`: `#~` (ou `gexp`) e `#$` (ou `ungexp`). Eles permitem *preparar código para execução posterior*.

  Por exemplo, você encontrará gexps em algumas definições de pacotes onde eles fornecem código a ser executado durante o processo de construção do pacote. Eles se parecem com isto:

```
(use-modules (guix gexp)           ; para que possamos escrever gexps
             (gnu packages base))  ; para 'coreutils'

;; Abaixo está uma expressão G representando código preparado.
#~(begin
    ;; Invoca 'ls' do pacote definido pela variável 'coreutils'.
    (system* #$(file-append coreutils "/bin/ls") "-l")

    ;; Crie o diretório de saída deste pacote.
    (mkdir #$output))
```

  Veja Seção "Expressões-G" em *Manual de Referência do GNU Guix*, para saber mais sobre gexps.

- Múltiplas variáveis podem ser nomeadas localmente com `let` (veja Seção "Local Bindings" em *Manual de referência do GNU Guile*):

```
(define x 10)
(let ((x 2)
      (y 3))
  (list x y))
⇒ (2 3)

x
⇒ 10

y
error  No procedimento module-lookup: Variável não vinculada: y
```

  Use `let*` para permitir que declarações de variáveis posteriores se refiram a definições anteriores.

```
(let* ((x 2)
       (y (* x 3)))
  (list x y))
⇒ (2 6)
```

- *Palavras-chave* normalmente são usados para identificar os parâmetros nomeados de um procedimento. Eles são prefixados por `#:` (hash, dois pontos) seguido por caracteres alfanuméricos: `#:like-this`. Veja Seção "Keywords" em *Manual de referência do GNU Guile*.
- A porcentagem `%` normalmente é usada para variáveis globais somente-leitura no estágio de construção. Observe que é apenas uma convenção, como `_` em C. Scheme trata `%` exatamente da mesma forma que qualquer outra letra.
- Os módulos são criados com `define-module` (veja Seção "Creating Guile Modules" em *Manual de referência do GNU Guile*). Por exemplo

```
(define-module (guix build-system ruby)
  #:use-module (guix store)
  #:export (ruby-build
            ruby-build-system))
```

  define o módulo `guix build-system ruby` que deve estar localizado em `guix/build-system/ruby.scm` em algum lugar no caminho de carregamento do Guile. Depende do módulo `(guix store)` e exporta duas variáveis, `ruby-build` e `ruby-build-system`.

  Veja Seção "Módulos de pacote" em *GNU Guix Reference Manual*, para informações sobre módulos que definem pacotes.

  Indo além: Scheme é uma linguagem que tem sido amplamente utilizada para ensinar programação e você encontrará muito material usando-a como veículo. Aqui está uma seleção de documentos para saber mais sobre o Scheme:

  - *A Scheme Primer* (`https://spritely.institute/static/papers/scheme-primer.html`), por Christine Lemmer-Webber e o Spritely Institute.
  - *Scheme at a Glance* (`http://www.troubleshooters.com/codecorn/scheme_guile/hello.htm`), por Steve Litt.
  - *Structure and Interpretation of Computer Programs* (`https://sarabander.github.io/sicp/`), por Harold Abelson e Gerald Jay Sussman, com Julie Sussman. Coloquialmente conhecido como "SICP", este livro é uma referência.

    Você também pode instalá-lo e lê-lo em seu computador:

```
guix install sicp info-reader
info sicp
```

  Você encontrará mais livros, tutoriais e outros recursos em `https://schemers.org/`.

2 Empacotamento

Este capítulo é dedicado a ensinar como adicionar pacotes à coleção de pacotes que vem com o GNU Guix. Isso envolve escrever definições de pacotes no Guile Scheme, organizá-las em módulos de pacotes e construí-las.

2.1 Tutorial sobre empacotamento

GNU Guix se destaca como o gerenciador de pacotes *hackeável*, principalmente porque usa GNU Guile (`https://www.gnu.org/software/guile/`), uma poderosa linguagem de programação de alto nível, um dos dialetos Scheme (`https://pt.wikipedia.org/wiki/Scheme`) da família Lisp (`https://pt.wikipedia.org/wiki/Lisp`).

As definições de pacotes também são escritas em Scheme, o que capacita o Guix de maneiras muito exclusivas, ao contrário da maioria dos outros gerenciadores de pacotes que usam shell scripts ou linguagens simples.

- Utilize funções, estruturas, macros e toda a expressividade do Scheme para definições de seus pacotes.
- A herança facilita a personalização de um pacote, herdando-o e modificando apenas o que é necessário.
- Processamento em lote: toda a coleção de pacotes pode ser analisada, filtrada e processada. Construindo um servidor headless com todas as interfaces gráficas removidas? É possível. Quer reconstruir tudo, desde o código-fonte usando sinalizadores específicos de otimização do compilador? Passe o argumento `#:make-flags "..."` para a lista de pacotes. Não seria exagero pensar em Gentoo USE flags (`https://wiki.gentoo.org/wiki/USE_flag`) aqui, mas isso vai ainda mais longe: as mudanças não precisam ser pensadas de antemão pelo empacotador, elas podem ser *programadas* pelo usuário!

O tutorial a seguir cobre todos os fundamentos da criação de pacotes com Guix. Não pressupõe muito conhecimento do sistema Guix nem da linguagem Lisp. Espera-se apenas que o leitor esteja familiarizado com a linha de comando e tenha alguns conhecimentos básicos de programação.

2.1.1 Um pacote "Hello World"

A seção "Definindo Pacotes" do manual apresenta os fundamentos do empacotamento Guix (veja Seção "Definindo pacotes" em *Manual de Referência do GNU Guix*). Na seção seguinte, revisaremos parcialmente esses princípios básicos novamente.

GNU Hello é um projeto fictício que serve como exemplo idiomático para empacotamento. Ele usa o sistema de compilação GNU (`./configure && make && make install`). Guix já fornece uma definição de pacote que é um exemplo perfeito para começar. Você pode consultar sua declaração com `guix edit hello` na linha de comando. Vamos ver como fica:

```
(define-public hello
  (package
    (name "hello")
    (version "2.10")
    (source (origin
```

```
                  (method url-fetch)
                  (uri (string-append "mirror://gnu/hello/hello-" version
                                      ".tar.gz"))
                  (sha256
                   (base32
                    "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
        (build-system gnu-build-system)
        (synopsis "Hello, GNU world: An example GNU package")
        (description
         "GNU Hello prints the message \"Hello, world!\" and then exits.  It
    serves as an example of standard GNU coding practices.  As such, it supports
    command-line arguments, multiple languages, and so on.")
        (home-page "https://www.gnu.org/software/hello/")
        (license gpl3+)))
```

Como você pode ver, a maior parte é bastante simples. Mas vamos revisar os campos juntos:

'`name`' O nome do projeto. Usando as convenções do Scheme, preferimos mantê-lo em minúsculas, sem sublinhado e usando palavras separadas por traços.

'`source`' Este campo contém uma descrição da origem do código-fonte. O registro `origin` contém estes campos:

1. O método, aqui `url-fetch` para download via HTTP/FTP, mas outros métodos existem, como `git-fetch` para repositórios Git.
2. O URI, que normalmente é alguma `https://` localização para `url-fetch`. Aqui o especial 'mirror://gnu' refere-se a um conjunto de locais bem conhecidos, todos os quais podem ser usados pelo Guix para buscar a fonte, caso alguns deles falhem.
3. A soma de verificação `sha256` do arquivo solicitado. Isto é essencial para garantir que a fonte não está corrompida. Observe que o Guix funciona com strings base32, daí a chamada para a função `base32`.

'`build-system`'
É aqui que o poder de abstração fornecido pela linguagem Scheme realmente brilha: neste caso, o `gnu-build-system` abstrai as famosas invocações de shell `./configure && make && make install`. Outros sistemas de compilação incluem o `trivial-build-system` que não faz nada e exige que o empacotador programe todas as etapas de compilação, o `python-build-system`, o `emacs-build- system` e muito mais (veja Seção "Sistemas de compilação" em *Manual de Referência do GNU Guix*).

'`synopsis`'
Deve ser um resumo conciso do que o pacote faz. Para muitos pacotes, uma etiqueta da página inicial do projeto pode ser usado como sinopse.

'`description`'
Assim como na sinopse, não há problema em reutilizar a descrição do projeto na página inicial. Observe que o Guix usa a sintaxe de Texinfo.

‘página inicial’
: Use HTTPS, se disponível.

‘license’
: Consulte `guix/licenses.scm` na fonte do projeto para obter uma lista completa de licenças disponíveis.

É hora de construir nosso primeiro pacote! Nada sofisticado aqui por enquanto: vamos nos ater a um `my-hello` fictício, uma cópia da declaração acima.

Tal como acontece com o ritualístico “Hello World” ensinado com a maioria das linguagens de programação, esta será possivelmente a abordagem mais “manual”. Trabalharemos em uma configuração ideal mais tarde; por enquanto seguiremos o caminho mais simples.

Salve o seguinte em um arquivo `my-hello.scm`.

```
(use-modules (guix packages)
             (guix download)
             (guix build-system gnu)
             (guix licenses))

(package
  (name "my-hello")
  (version "2.10")
  (source (origin
            (method url-fetch)
            (uri (string-append "mirror://gnu/hello/hello-" version
                                ".tar.gz"))
            (sha256
             (base32
              "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
  (build-system gnu-build-system)
  (synopsis "Hello, Guix world: An example custom Guix package")
  (description
   "GNU Hello prints the message \"Hello, world!\" and then exits.  It
serves as an example of standard GNU coding practices.  As such, it supports
command-line arguments, multiple languages, and so on.")
  (home-page "https://www.gnu.org/software/hello/")
  (license gpl3+))
```

Explicaremos o código extra em um momento.

Sinta-se à vontade para brincar com os diferentes valores dos vários campos. Se você alterar a fonte, precisará atualizar a soma de verificação. Na verdade, Guix se recusa a construir qualquer coisa se a soma de verificação fornecida não corresponder à soma de verificação calculada do código-fonte. Para obter a soma de verificação correta da declaração do pacote, precisamos baixar o código-fonte, calcular a soma de verificação sha256 e convertê-la para base32.

Felizmente, o Guix pode automatizar essa tarefa para nós; tudo o que precisamos é fornecer o URI:

```
$ guix download mirror://gnu/hello/hello-2.10.tar.gz
```

```
Starting download of /tmp/guix-file.JLYgL7
From https://ftpmirror.gnu.org/gnu/hello/hello-2.10.tar.gz...
following redirection to `https://mirror.ibcp.fr/pub/gnu/hello/hello-2.10.tar.gz'...
 ...10.tar.gz  709KiB                                  2.5MiB/s 00:00 [################
/gnu/store/hbdalsf5lpf01x4dcknwx6xbn6n5km6k-hello-2.10.tar.gz
0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i
```

Neste caso específico a saída nos informa qual espelho foi escolhido. Se o resultado do comando acima não for o mesmo do trecho acima, atualize sua declaração `my-hello` de acordo.

Observe que os tarballs do pacote GNU vêm com uma assinatura OpenPGP, então você definitivamente deve verificar a assinatura deste tarball com 'gpg' para autenticá-lo antes de prosseguir:

```
$ guix download mirror://gnu/hello/hello-2.10.tar.gz.sig

Iniciando download de /tmp/guix-file.03tFfb
De https://ftpmirror.gnu.org/gnu/hello/hello-2.10.tar.gz.sig...
seguindo redirecionamento para `https://ftp.igh.cnrs.fr/pub/gnu/hello/hello-2.10.tar.g
....tar.gz.sig 819B 1,2 MiB/s 00:00 [##################] 100,0%
/gnu/store/rzs8wba9ka7grrmgcpfyxvs58mly0sx6-hello-2.10.tar.gz.sig
0q0v86n3y38z17rl146gdakw9xc4mcscpk8dscs412j22glrv9jf
$ gpg --verify /gnu/store/rzs8wba9ka7grrmgcpfyxvs58mly0sx6-hello-2.10.tar.gz.sig /gnu/
gpg: Assinatura feita em dom 16 nov 2014 01:08:37 PM CET
gpg: usando RSA chave A9553245FDE9B739
gpg: Boa assinatura de "Sami Kerola <kerolasa@iki.fi>" [desconhecido]
gpg: também conhecido como "Sami Kerola (http://www.iki.fi/kerolasa/) <kerolasa@iki.fi
gpg: AVISO: Esta chave não é certificada com uma assinatura confiável!
gpg: Não há indicação de que a assinatura pertença ao proprietário.
Impressão digital da chave primária: 8ED3 96E3 7E38 D471 A005 30D3 A955 3245 FDE9 B739
```

Você pode então correr alegremente

```
$ guix package --install-from-file=my-hello.scm
```

Agora você deve ter `my-hello` em seu perfil!

```
$ guix package --list-installed=my-hello
my-hello 2.10 out
/gnu/store/f1db2mfm8syb8qvc357c53slbvf1g9m9-my-hello-2.10
```

Fomos o mais longe que pudemos sem qualquer conhecimento do Scheme. Antes de passar para pacotes mais complexos, agora é o momento certo para aprimorar seus conhecimentos sobre o Scheme. Veja Seção 1.1 [Um curso intensivo de Scheme], Página 1, para se atualizar.

2.1.2 Configuração

No restante deste capítulo contaremos com alguns conhecimentos básicos de programação de Scheme. Agora vamos detalhar as diferentes configurações possíveis para trabalhar em pacotes Guix.

Existem várias maneiras de configurar um ambiente de empacotamento Guix.

Recomendamos que você trabalhe diretamente no checkout do código-fonte do Guix, pois facilita a contribuição de todos para o projeto.

Mas primeiro, vamos examinar outras possibilidades.

2.1.2.1 Arquivo local

Isto é o que fizemos anteriormente com '`my-hello`'. Com os princípios básicos do Scheme que cobrimos, agora podemos explicar os principais pedaços. Conforme declarado em `guix package --help`:

```
-f, --install-from-file=ARQUIVO
                        instala o pacote para o qual o código
                        dentro do ARQUIVO avalia
```

Assim, a última expressão *deve* retornar um pacote, que é o caso do nosso exemplo anterior.

A expressão `use-modules` informa quais módulos precisamos no arquivo. Módulos são uma coleção de valores e procedimentos. Eles são comumente chamados de "bibliotecas" ou "pacotes" em outras linguagens de programação.

2.1.2.2 Canais

Guix e sua coleção de pacotes podem ser estendidos através de *canais*. Um canal é um repositório Git, público ou não, contendo arquivos `.scm` que fornecem pacotes (veja Seção "Definindo pacotes" em *Manual de Referência do GNU Guix*) ou serviços (veja Seção "Definindo serviços" em *Manual de Referência GNU Guix*).

Como você criaria um canal? Primeiro, crie um diretório que conterá seus arquivos `.scm`, digamos `~/my-channel`:

```
mkdir ~/my-channel
```

Suponha que você queira adicionar o pacote '`my-hello`' que vimos anteriormente; primeiro precisa de alguns ajustes:

```
(define-module (my-hello)
  #:use-module (guix licenses)
  #:use-module (guix packages)
  #:use-module (guix build-system gnu)
  #:use-module (guix download))

(define-public my-hello
  (package
    (name "my-hello")
    (version "2.10")
    (source (origin
              (method url-fetch)
              (uri (string-append "mirror://gnu/hello/hello-" version
                                  ".tar.gz"))
              (sha256
               (base32
                "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
    (build-system gnu-build-system)
    (synopsis "Olá, mundo Guix: Um exemplo de pacote Guix personalizado")
    (description
```

```
     "GNU Hello imprime a mensagem \"Olá, mundo!\" e então sai. Ele
serve como um exemplo de práticas de codificação GNU padrão. Como tal, ele suporta
argumentos de linha de comando, vários idiomas e assim por diante.")
    (home-page "https://www.gnu.org/software/hello/")
    (license gpl3+)))
```

Observe que atribuímos o valor do pacote a um nome de variável exportado com `define-public`. Isso é efetivamente atribuir o pacote à variável `my-hello` para que ele possa ser referenciado, entre outras coisas, como dependência de outros pacotes.

Se você usar `guix package --install-from-file=my-hello.scm` no arquivo acima, ele falhará porque a última expressão, `define-public`, não retorna um pacote. Mesmo assim, se você quiser usar `define-public` neste caso de uso, certifique-se de que o arquivo termine com uma avaliação de `my-hello`:

```
;; ...
(define-public my-hello
  ;; ...
  )

my-hello
```

Este último exemplo não é muito típico.

Agora, como você torna esse pacote visível para os comandos `guix` para poder testar seus pacotes? Você precisa adicionar o diretório ao caminho de pesquisa usando a opção de linha de comando `-L`, como nestes exemplos:

```
guix show -L ~/my-channel my-hello
guix build -L ~/my-channel my-hello
```

A etapa final é transformar `~/my-channel` em um canal real, disponibilizando sua coleção de pacotes perfeitamente *via* qualquer comando `guix`. Para fazer isso, primeiro você precisa torná-lo um repositório Git:

```
cd ~/my-channel
git init
git add my-hello.scm
git commit -m "Primeiro commit do meu canal."
```

E pronto, você tem um canal! A partir daí, você pode adicionar este canal à configuração do seu canal em `~/.config/guix/channels.scm` (veja Seção "Especificando canais adicionais" em *Manual de Referência do GNU Guix*); supondo que você mantenha seu canal local por enquanto, o `channels.scm` ficaria mais ou menos assim:

```
(append (list (channel
                (name 'my-channel)
                (url (string-append "file://" (getenv "HOME")
                                    "/my-channel"))))
        %default-channels)
```

Da próxima vez que você executar `guix pull`, seu canal será selecionado e os pacotes que ele definir estarão prontamente disponíveis para todos os comandos `guix`, mesmo se você não passar `-L`. O comando `guix description` mostrará que o Guix está, de fato, usando os canais `my-channel` e `guix`.

Veja Seção "Criando um canal" em *Manual de Referência do GNU Guix*, para detalhes.

2.1.2.3 Conferir o hacking direto

É recomendado trabalhar diretamente no projeto Guix: isso reduz o atrito quando chega a hora de enviar suas alterações ao upstream para permitir que a comunidade se beneficie de seu trabalho árduo!

Ao contrário da maioria das distribuições de software, o repositório Guix mantém em um só lugar as ferramentas (incluindo o gerenciador de pacotes) e as definições dos pacotes. Essa escolha foi feita para dar aos desenvolvedores a flexibilidade de modificar a API sem quebras, atualizando todos os pacotes ao mesmo tempo. Isto reduz a inércia do desenvolvimento.

Confira o repositório oficial Git (`https://git-scm.com/`):

```
$ git clone https://git.savannah.gnu.org/git/guix.git
```

No restante deste artigo, usamos '`$GUIX_CHECKOUT`' para nos referir ao local do checkout.

Siga as instruções no manual (veja Seção "Contribuindo" em *Manual de Referência do GNU Guix*) para configurar o ambiente do repositório.

Quando estiver pronto, você poderá usar as definições de pacote do ambiente do repositório.

Sinta-se à vontade para editar as definições de pacotes encontradas em '`$GUIX_CHECKOUT/gnu/packages`'.

O script '`$GUIX_CHECKOUT/pre-inst-env`' permite usar '`guix`' sobre a coleção de pacotes do repositório (veja Seção "Executando guix antes dele ser instalado" em *Manual de Referência do GNU Guix*) .

- Pesquisa de pacotes, como Ruby:

```
$ cd $GUIX_CHECKOUT
$ ./pre-inst-env guix package --list-available=ruby
   ruby    1.8.7-p374      out     gnu/packages/ruby.scm:119:2
   ruby    2.1.6   out     gnu/packages/ruby.scm:91:2
   ruby    2.2.2   out     gnu/packages/ruby.scm:39:2
```

- Construa um pacote, como Ruby versão 2.1:

```
$ ./pre-inst-env guix build --keep-failed ruby@2.1
/gnu/store/c13v73jxmj2nir2xjqaz5259zywsa9zi-ruby-2.1.6
```

- Instale-o em seu perfil de usuário:

```
$ ./pre-inst-env guix package --install ruby@2.1
```

- Verifique se há erros comuns:

```
$ ./pre-inst-env guix lint ruby@2.1
```

Guix se esforça para manter um alto padrão de empacotamento; ao contribuir para o projeto Guix, lembre-se de

- seguir o estilo de codificação (veja Seção "Estilo de código" em *Manual de Referência do GNU Guix*),
- e revise a lista de verificação do manual (veja Seção "Enviando patches" em *Manual de Referência do GNU Guix*).

Quando estiver satisfeito com o resultado, você pode enviar sua contribuição para torná-lo parte do Guix. Este processo também é detalhado no manual. (veja Seção "Contribuindo" em *Manual de Referência GNU Guix*)

É um esforço da comunidade, então quanto mais participar, melhor o Guix se torna!

2.1.3 Exemplo estendido

O exemplo "Hello World" acima é tão simples quanto parece. Os pacotes podem ser mais complexos do que isso e o Guix pode lidar com cenários mais avançados. Vejamos outro pacote mais sofisticado (ligeiramente modificado em relação à fonte):

```
(define-module (gnu packages version-control)
  #:use-module ((guix licenses) #:prefix license:)
  #:use-module (guix utils)
  #:use-module (guix packages)
  #:use-module (guix git-download)
  #:use-module (guix build-system cmake)
  #:use-module (gnu packages compression)
  #:use-module (gnu packages pkg-config)
  #:use-module (gnu packages python)
  #:use-module (gnu packages ssh)
  #:use-module (gnu packages tls)
  #:use-module (gnu packages web))

(define-public my-libgit2
  (let ((commit "e98d0a37c93574d2c6107bf7f31140b548c6a7bf")
        (revision "1"))
    (package
      (name "my-libgit2")
      (version (git-version "0.26.6" revision commit))
      (source (origin
                (method git-fetch)
                (uri (git-reference
                      (url "https://github.com/libgit2/libgit2/")
                      (commit commit)))
                (file-name (git-file-name name version))
                (sha256
                 (base32
                  "17pjvprmdrx4h6bb1hhc98w9qi6ki7yl57f090n9kbhswxqfs7s3"))
                (patches (search-patches "libgit2-mtime-0.patch"))
                (modules '((guix build utils)))
                ;; Remover software empacotado.
                (snippet '(delete-file-recursively "deps"))))
      (build-system cmake-build-system)
      (outputs '("out" "debug"))
      (arguments
       `(#:tests? #true                         ; Execute o conjunto de testes (este é
         #:configure-flags '("-DUSE_SHA1DC=ON") ; Detecção de colisão SHA-1
         #:phases
         (modify-phases %standard-phases
           (add-after 'unpack 'fix-hardcoded-paths
             (lambda _
               (substitute* "tests/repo/init.c"
```

```
                  (("#!/bin/sh") (string-append "#!" (which "sh"))))
                (substitute* "tests/clar/fs.h"
                  (("/bin/cp") (which "cp"))
                  (("/bin/rm") (which "rm")))))
            ;; Execute verificações de forma mais detalhada.
            (replace 'check
              (lambda* (#:key tests? #:allow-other-keys)
                (when tests?
                  (invoke "./libgit2_clar" "-v" "-Q"))))
            (add-after 'unpack 'make-files-writable-for-tests
              (lambda _ (for-each make-file-writable (find-files "."))))))
        (inputs
         (list libssh2 http-parser python-wrapper))
        (native-inputs
         (list pkg-config))
        (propagated-inputs
         ;; Essas duas bibliotecas estão em 'Requires.private' em libgit2.pc.
         (list openssl zlib))
        (home-page "https://libgit2.github.com/")
        (synopsis "Biblioteca que fornece métodos básicos do Git")
        (description
         "Libgit2 é uma implementação C pura e portátil dos métodos principais do Git
fornecida como uma biblioteca vinculável reentrante com uma API sólida, permitindo que
escreva aplicativos Git personalizados de velocidade nativa em qualquer linguagem com
        ;; GPLv2 com exceção de vinculação
        (license license:gpl2))))
```

(Nos casos em que você deseja ajustar apenas alguns campos de uma definição de pacote, você deve confiar na herança em vez de copiar e colar tudo. Veja abaixo.)

Vamos discutir esses campos em profundidade.

2.1.3.1 Método `git-fetch`

Ao contrário do método `url-fetch`, `git-fetch` espera um `git-reference` que usa um repositório Git e um commit. O commit pode ser qualquer referência do Git, como tags, portanto, se `version` estiver marcado, ele poderá ser usado diretamente. Às vezes, a tag é prefixada com `v`; nesse caso, você usaria `(commit (string-append "v" version))`.

Para garantir que o código-fonte do repositório Git seja armazenado em um diretório com um nome descritivo, usamos `(nomedo- arquivo (nome-do-arquivo-git versão))`.

O procedimento `git-version` pode ser usado para derivar a versão ao empacotar programas para um commit específico, seguindo as diretrizes do contribuidor Guix (veja Seção "Números de versão" em *Manual de Referência do GNU Guix*).

Como obter o hash `sha256` que está aí, você pergunta? Invocando `guix hash` em um checkout do commit desejado, da seguinte forma:

```
git clone https://github.com/libgit2/libgit2/
cd libgit2
git checkout v0.26.6
```

```
guix hash -rx .
```

`guix hash -rx` calcula um hash SHA256 em todo o diretório, excluindo o subdiretório `.git` (veja Seção "Invocando guix hash" em *Manual de Referência do GNU Guix*).

No futuro, `guix download` será capaz de executar essas etapas para você, assim como faz para downloads regulares.

2.1.3.2 Trechos

Os trechos ("Snippets") são códigos de Scheme citados (ou seja, não avaliados) que são um meio de corrigir a fonte. Eles são uma alternativa Guix-y aos arquivos `.patch` tradicionais. Por causa da citação, o código só é avaliado quando passado para o daemon Guix para construção. Pode haver quantos trechos forem necessários.

Os snippets podem precisar de módulos Guile adicionais que podem ser importados do campo `modules`.

2.1.3.3 Entradas

Existem 3 tipos de entradas diferentes. Resumidamente:

native-inputs
: Necessário para construção, mas não para tempo de execução - instalar um pacote por meio de um substituto não instalará essas entradas.

inputs
: Instalado no armazém mas não no perfil, além de estar presente na hora da construção.

propagated-inputs
: Instalado no armazém e no perfil, além de estar presente na hora da construção.

Veja Seção "Referência do package" em *Manual de referência GNU Guix* para mais detalhes.

A distinção entre as diversas entradas é importante: se uma dependência puder ser tratada como uma *input* em vez de uma *propagated input*, isso deverá ser feito, caso contrário ela "poluirá" o perfil do usuário sem nenhuma boa razão.

Por exemplo, um usuário que instala um programa gráfico que depende de uma ferramenta de linha de comando pode estar interessado apenas na parte gráfica, portanto não há necessidade de forçar a ferramenta de linha de comando no perfil do usuário. A dependência é uma preocupação do pacote, não do usuário. *Inputs* tornam possível lidar com dependências sem incomodar o usuário, adicionando arquivos executáveis (ou bibliotecas) indesejados ao seu perfil.

O mesmo vale para *native-inputs*: depois que o programa é instalado, as dependências em tempo de construção podem ser coletadas como lixo com segurança. Também importa quando um substituto está disponível, caso em que apenas *inputs* e *propagated inputs* serão obtidos: os *native inputs* não são necessários para instalar um pacote de um substituto.

> **Nota:** Você pode ver aqui e ali trechos onde as entradas do pacote são escritas de maneira bem diferente, assim:
>
> ```
> ;; O "estilo antigo" para entradas.
> (inputs
>  `(("libssh2" ,libssh2)
> ```

```
("http-parser" ,http-parser)
("python" ,python-wrapper)))
```

Este é o "estilo antigo", onde cada entrada na lista recebe explicitamente um rótulo (uma string). Ainda é compatível, mas recomendamos usar o estilo acima. Veja Seção "Referência do package" em *Manual de Referência do GNU Guix*, para mais informações.

2.1.3.4 Saídas

Assim como um pacote pode ter múltiplas entradas, ele também pode produzir múltiplas saídas.

Cada saída corresponde a um diretório separado no armazém.

O usuário pode escolher qual saída instalar; isso é útil para economizar espaço ou evitar poluir o perfil do usuário com executáveis ou bibliotecas indesejadas.

A separação de saída é opcional. Quando o campo `outputs` é omitido, a saída padrão e única (o pacote completo) é referida como `"out"`.

Nomes de saída separados típicos incluem `debug` e `doc`.

É aconselhável separar as saídas apenas quando você mostrar que vale a pena: se o tamanho da saída for significativo (compare com `guix size`) ou caso o pacote seja modular.

2.1.3.5 Argumentos do sistema de compilação

O `arguments` é uma lista de valores de palavras-chave usadas para configurar o processo de construção.

O argumento mais simples `#:tests?` pode ser usado para desabilitar o conjunto de testes ao construir o pacote. Isso é útil principalmente quando o pacote não apresenta nenhum conjunto de testes. É altamente recomendável manter o conjunto de testes ativado, se houver.

Outro argumento comum é `:make-flags`, que especifica uma lista de sinalizadores a serem acrescentados ao executar o make, como faria na linha de comando. Por exemplo, os seguintes sinalizadores

```
#:make-flags (list (string-append "prefix=" (assoc-ref %outputs "out"))
                   "CC=gcc")
```

traduzir para

```
$ make CC=gcc prefix=/gnu/store/...-<out>
```

Isso define o compilador C para `gcc` e a variável `prefix` (o diretório de instalação no jargão Make) para `(assoc-ref %outputs "out")`, que é um estágio de construção global variável apontando para o diretório de destino no armazém (algo como `/gnu/store/...-my-libgit2-20180408`).

Da mesma forma, é possível definir os sinalizadores de configuração:

```
#:configure-flags '("-DUSE_SHA1DC=ON")
```

A variável `%build-inputs` também é gerada no escopo. Ela é uma tabela de associações que mapeia os nomes de entrada para seus diretórios de armazém.

A palavra-chave `phases` lista as etapas sequenciais do sistema de compilação. Normalmente as fases incluem `unpack`, `configure`, `build`, `install` e `check`. Para saber mais

sobre essas fases, você precisa definir a definição apropriada do sistema de compilação em '`$GUIX_CHECKOUT/guix/build/gnu-build-system.scm`':

```
(define %standard-phases
  ;; Fases de construção padrão, como uma lista de pares de símbolos/procedimentos.
  (let-syntax ((phases (syntax-rules ()
                         ((_ p ...) `((p . ,p) ...)))))
    (phases set-SOURCE-DATE-EPOCH set-paths install-locale unpack
            bootstrap
            patch-usr-bin-file
            patch-source-shebangs configure patch-generated-file-shebangs
            build check install
            patch-shebangs strip
            validate-runpath
            validate-documentation-location
            delete-info-dir-file
            patch-dot-desktop-files
            install-license-files
            reset-gzip-timestamps
            compress-documentation)))
```

Ou do REPL:

```
(add-to-load-path "/path/to/guix/checkout")
,use (guix build gnu-build-system)
(map first %standard-phases)
⇒ (set-SOURCE-DATE-EPOCH set-paths install-locale unpack bootstrap patch-usr-bin-file
```

Se quiser saber mais sobre o que acontece nessas fases, consulte os procedimentos associados.

Por exemplo, no momento em que este livro foi escrito, a definição de `unpack` para o sistema de compilação GNU era:

```
(define* (unpack #:key source #:allow-other-keys)
  "Descompacte SOURCE no diretório de trabalho e altere o diretório dentro do
source. Quando SOURCE for um diretório, copie-o em um subdiretório do atual
diretório de trabalho."
  (if (file-is-directory? source)
      (begin
        (mkdir "source")
        (chdir "source")

        ;; Preservar carimbos de data/hora (definidos para a Época) na
        ;; árvore copiada para que as coisas funcionem de forma
        ;; determinística.
        (copy-recursively source "."
                          #:keep-mtime? #true))
      (begin
        (if (string-suffix? ".zip" source)
            (invoke "unzip" source)
```

```
            (invoke "tar" "xvf" source))
          (chdir (first-subdirectory ".")))))
    #true)
```

Observe a chamada `chdir`: ela altera o diretório de trabalho para onde a fonte foi descompactada. Assim, cada fase após o `unpack` usará a fonte como diretório de trabalho, e é por isso que podemos trabalhar diretamente nos arquivos de origem. Isto é, a menos que uma fase posterior mude o diretório de trabalho para outro.

Modificamos a lista de `%standard-phases` do sistema de construção com a macro `modify-phases` conforme a lista de modificações especificadas, que pode ter os seguintes formatos:

- `(add-before` *`phase new-phase procedure`*`)`: Execute *procedure* chamado *new-phase* antes de *phase*.
- `(add-after` *`phase new-phase procedure`*`)`: O mesmo, mas depois.
- `(replace` *`phase procedure`*`)`.
- `(delete` *`phase`*`)`.

O *procedure* suporta os argumentos de palavra-chave `inputs` e `outputs`. Cada entrada (seja *native*, *propagated* ou não) e diretório de saída são referenciados por seus nomes nessas variáveis. Assim `(assoc-ref outputs "out")` é o diretório de armazém da saída principal do pacote. Um procedimento de fase pode ser assim:

```
(lambda* (#:key inputs outputs #:allow-other-keys)
  (let ((bash-directory (assoc-ref inputs "bash"))
        (output-directory (assoc-ref outputs "out"))
        (doc-directory (assoc-ref outputs "doc")))
    ;; ...
    #true))
```

O procedimento deve retornar `#true` em caso de sucesso. É frágil confiar no valor de retorno da última expressão usada para ajustar a fase porque não há garantia de que seria um `#true`. Daí o `#true` final para garantir que o valor correto seja retornado em caso de sucesso.

2.1.3.6 Preparação de código

O leitor astuto deve ter notado a sintaxe de quase aspas e vírgula no campo do argumento. Na verdade, o código de construção na declaração do pacote não deve ser avaliado no lado do cliente, mas apenas quando passado para o daemon Guix. Esse mecanismo de passagem de código entre dois processos em execução é chamado preparação de código (`https://arxiv.org/abs/1709.00833`).

2.1.3.7 Funções utilitárias

Ao personalizar `phases`, muitas vezes precisamos escrever código que imite as invocações equivalentes do sistema (`make`, `mkdir`, `cp`, etc.) comumente usado durante “ Instalações regulares no estilo Unix”.

Alguns como `chmod` são nativos do Guile. Veja *manual de referência do Guile* para obter uma lista completa.

Guix fornece funções auxiliares adicionais que são especialmente úteis no contexto de gerenciamento de pacotes.

Algumas dessas funções podem ser encontradas em '`$GUIX_CHECKOUT/guix/guix/build/utils.scm`'. A maioria delas reflete o comportamento dos comandos tradicionais do sistema Unix:

`which` Como o comando do sistema '`which`'.

`find-files`
Semelhante ao comando do sistema '`find`'.

`mkdir-p` Como '`mkdir -p`', que cria todos os diretórios conforme necessário.

`install-file`
Semelhante a '`install`' ao instalar um arquivo em um diretório (possivelmente inexistente). Guile tem `copy-file` que funciona como '`cp`'.

`copy-recursively`
Como '`cp -r`'.

`delete-file-recursively`
Como '`rm -rf`'.

`invoke` Executar um executável. Deve ser usado em vez de `system*`.

`with-directory-excursion`
Execute o corpo em um diretório de trabalho diferente e restaure o diretório de trabalho anterior.

`substitute*`
Uma função do tipo "`sed`".

Veja Seção "Construir utilitários" em *Manual de Referência GNU Guix*, para obter mais informações sobre esses utilitários.

2.1.3.8 Prefixo do módulo

A licença em nosso último exemplo precisa de um prefixo: isso se deve à forma como o módulo `license` foi importado no pacote, como `#:use-module ((guix Licenses) #:prefix License:)`. O mecanismo de importação do módulo Guile (veja Seção "Using Guile Modules" em *manual de referência do Guile*) dá ao usuário controle total sobre o namespace: isso é necessário para evitar conflitos entre, digamos, a variável '`zlib`' de '`licenses.scm`' (um valor *license*) e a variável '`zlib`' de '`compression.scm`' (um valor *package*).

2.1.4 Outros sistemas de construção

O que vimos até agora cobre a maioria dos pacotes que usam um sistema de compilação diferente do `trivial-build-system`. Este último não automatiza nada e deixa você construir tudo manualmente. Isso pode ser mais exigente e não abordaremos isso aqui por enquanto, mas felizmente raramente é necessário recorrer a este sistema.

Para outros sistemas de construção, como ASDF, Emacs, Perl, Ruby e muitos outros, o processo é muito semelhante ao sistema de construção GNU, exceto por alguns argumentos especializados.

Veja Seção “Sistemas de compilação” em *Manual de Referência GNU Guix*, para obter mais informações sobre sistemas de construção, ou verifique o código-fonte em ‘`$GUIX_CHECKOUT/guix/build`’ e ‘`$GUIX_CHECKOUT/guix/build -sistema`’ diretórios.

2.1.5 Definição de pacote programável e automatizada

Não podemos repetir o suficiente: ter uma linguagem de programação completa em mãos nos capacita de maneiras que vão muito além do gerenciamento tradicional de pacotes.

Vamos ilustrar isso com alguns recursos incríveis do Guix!

2.1.5.1 Importadores recursivos

Você pode achar alguns sistemas de compilação bons o suficiente para que haja pouco a fazer para escrever um pacote, a ponto de se tornar repetitivo e tedioso depois de um tempo. Uma *razão de ser* dos computadores é substituir os seres humanos nessas tarefas chatas. Então, vamos dizer ao Guix para fazer isso para nós e criar a definição de pacote de um pacote R do CRAN (a saída é cortada para ser concisa):

```
$ guix import cran --recursive walrus

(define-public r-mc2d
    ; ...
    (license gpl2+)))

(define-public r-jmvcore
    ; ...
    (license gpl2+)))

(define-public r-wrs2
    ; ...
    (license gpl3)))

(define-public r-walrus
  (package
    (name "r-walrus")
    (version "1.0.3")
    (source
      (origin
        (method url-fetch)
        (uri (cran-uri "walrus" version))
        (sha256
          (base32
            "1nk2glcvy4hyksl5ipq2mz8jy4fss90hx6cq98m3w96kzjni6jjj"))))
    (build-system r-build-system)
    (propagated-inputs
      (list r-ggplot2 r-jmvcore r-r6 r-wrs2))
    (home-page "https://github.com/jamovi/walrus")
    (synopsis "Métodos Estatísticos Robustos")
    (description
```

```
      "Este pacote fornece uma caixa de ferramentas de testes estatísticos robustos
comuns, incluindo descritivos robustos, testes t robustos e ANOVA robusto.
Ele também está disponível como um módulo para 'jamovi' (veja
<https://www.jamovi.org> para mais informações). O Walrus é baseado no
pacote WRS2 de Patrick Mair, que por sua vez é baseado nos scripts e
trabalho de Rand Wilcox. Essas análises são descritas em profundidade no livro
'Introdução à Estimativa Robusta e Teste de Hipóteses'.")
    (license gpl3)))
```

O importador recursivo não importará pacotes para os quais o Guix já possui definições de pacote, exceto o primeiro.

Nem todos os aplicativos podem ser empacotados dessa forma, apenas aqueles que dependem de um número selecionado de sistemas suportados. Leia sobre a lista completa de importadores na seção de importação de guix do manual (veja Seção "Invocando guix import" em *Manual de Referência do GNU Guix*).

2.1.5.2 Atualização automática

O Guix pode ser inteligente o suficiente para verificar atualizações nos sistemas que conhece. Ele pode relatar definições de pacotes desatualizadas com

```
$ guix refresh hello
```

Na maioria dos casos, atualizar um pacote para uma versão mais recente requer pouco mais do que alterar o número da versão e a soma de verificação. Guix também pode fazer isso automaticamente:

```
$ guix refresh hello --update
```

2.1.5.3 Herança

Se você começou a navegar pelas definições de pacotes existentes, deve ter notado que um número significativo deles possui um campo `inherit`:

```
(define-public adwaita-icon-theme
  (package (inherit gnome-icon-theme)
    (name "adwaita-icon-theme")
    (version "3.26.1")
    (source (origin
              (method url-fetch)
              (uri (string-append "mirror://gnome/sources/" name "/"
                                  (version-major+minor version) "/"
                                  name "-" version ".tar.xz"))
              (sha256
               (base32
                "17fpahgh5dyckgz7rwqvzgnhx53cx9kr2xw0szprc6bnqy977fi8"))))
    (native-inputs (list `(,gtk+ "bin")))))
```

Todos os campos não especificados são herdados do pacote pai. Isto é muito conveniente para criar pacotes alternativos, por exemplo, com diferentes fontes, versões ou opções de compilação.

2.1.6 Obtendo ajuda

Infelizmente, alguns aplicativos podem ser difíceis de empacotar. Às vezes, eles precisam de um patch para funcionar com a hierarquia do sistema de arquivos não padrão imposta pelo armazém. Às vezes, os testes não funcionam corretamente. (Eles podem ser ignorados, mas isso não é recomendado.) Outras vezes, o pacote resultante não será reproduzível.

Se você estiver emperrado, incapaz de descobrir como resolver qualquer tipo de problema de empacotamento, não hesite em pedir ajuda à comunidade.

Consulte página inicial do Guix (`https://www.gnu.org/software/guix/contact/`) para obter informações sobre listas de discussão, IRC, etc.

2.1.7 Conclusão

Este tutorial foi uma amostra do sofisticado gerenciamento de pacotes que o Guix possui. Neste ponto, restringimos principalmente esta introdução ao `gnu-build-system`, que é uma camada de abstração central na qual se baseiam abstrações mais avançadas.

Para onde vamos daqui? Em seguida, devemos dissecar as entranhas do sistema de construção, removendo todas as abstrações, usando o `trivial-build-system`: isso deve nos dar uma compreensão completa do processo antes de investigar algumas técnicas de empacotamento mais avançadas e casos extremos.

Outros recursos que valem a pena explorar são os recursos interativos de edição e depuração do Guix fornecidos pelo Guile REPL.

Esses recursos sofisticados são totalmente opcionais e podem esperar; agora é um bom momento para fazer uma pausa bem merecida. Com o que apresentamos aqui você deve estar bem preparado para empacotar muitos programas. Você pode começar imediatamente e esperamos ver suas contribuições em breve!

2.1.8 Referências

- O referência do pacote no manual (`https://www.gnu.org/software/guix/manual/en/html_node/Defining-Packages.html`)
- guia de hacking de Pjotr para GNU Guix (`https://gitlab.com/pjotrp/guix-notes/blob/master/HACKING.org`)
- "GNU Guix: Pacote sem Scheme!" (`https://www.gnu.org/software/guix/guix-ghm-andreas-20130823.pdf`), por Andreas Enge

3 Configuração do sistema

Guix oferece uma linguagem flexível para configurar declarativamente seu sistema Guix. Essa flexibilidade às vezes pode ser esmagadora. O objetivo deste capítulo é demonstrar alguns conceitos avançados de configuração.

veja Seção "Configuração do sistema" em *Manual de referência do GNU Guix* para uma referência completa.

3.1 Login automático em um TTY específico

Embora o manual do Guix explique o login automático de um usuário para *todas* TTYs (veja Seção "auto-login to TTY" em *Manual de Referência do GNU Guix*), alguns podem preferir uma situação em que um usuário está logado em um TTY com os outros TTYs configurados para fazer login com usuários diferentes ou com nenhum. Observe que é possível fazer login automático de um usuário em qualquer TTY, mas geralmente é aconselhável evitar `tty1`, que, por padrão, é usado para registrar avisos e erros.

Aqui está como se pode configurar o login automático para um usuário em um tty:

```
(define (auto-login-to-tty config tty user)
  (if (string=? tty (mingetty-configuration-tty config))
        (mingetty-configuration
         (inherit config)
         (auto-login user))
        config))

(define %my-services
  (modify-services %base-services
    ;; ...
    (mingetty-service-type config =>
                           (auto-login-to-tty
                            config "tty3" "alice"))))

(operating-system
  ;; ...
  (services %my-services))
```

Pode-se também `compose` (veja Seção "Higher-Order Functions" em *guile*) `auto-login-to-tty` para fazer login de vários usuários em vários ttys.

Finalmente, aqui está uma nota de cautela. Configurar o login automático em um TTY significa que qualquer pessoa pode ligar seu computador e executar comandos como seu usuário normal. No entanto, se você tiver uma partição raiz criptografada e, portanto, já precisar inserir uma senha quando o sistema inicializar, o login automático pode ser uma opção conveniente.

3.2 Customizando o Kernel

Guix é, em sua essência, uma distribuição baseada em código-fonte com substitutos (veja Seção "Substitutos" em *Manual de Referência do GNU Guix*) e, como tal, construir pacotes

a partir de seu código-fonte é uma parte esperada das instalações e atualizações regulares de pacotes. Dado este ponto de partida, faz sentido que sejam feitos esforços para reduzir a quantidade de tempo gasto na compilação de pacotes, e as recentes mudanças e atualizações na construção e distribuição de substitutos continuam a ser um tópico de discussão dentro do Guix.

O kernel, embora não exija uma superabundância de RAM para ser construído, leva muito tempo em uma máquina média. A configuração oficial do kernel, como é o caso de muitas distribuições GNU/Linux, erra pelo lado da inclusão, e é isso que realmente faz com que a construção demore tanto tempo quando o kernel é compilado a partir do código-fonte.

O kernel do Linux, entretanto, também pode ser descrito apenas como um pacote antigo normal e, como tal, pode ser personalizado como qualquer outro pacote. O procedimento é um pouco diferente, embora isso se deva principalmente à natureza de como a definição do pacote é escrita.

A definição do pacote do kernel `linux-libre` é na verdade um procedimento que cria um pacote.

```
(define* (make-linux-libre* versão gnu-revision fonte sistemas-suportados
                            #:key
                            (extra-version #f)
                            ;; Uma função que pega um arch e uma variante.
                            ;; Veja kernel-config para um exemplo.
                            (configuration-file #f)
                            (defconfig "defconfig")
                            (extra-options %default-extra-linux-options))
  ...)
```

O pacote `linux-libre` atual é para a série 5.15.x e é declarado assim:

```
(define-public linux-libre-5.15
  (make-linux-libre* linux-libre-5.15-version
                     linux-libre-5.15-gnu-revision
                     linux-libre-5.15-source
                     '("x86_64-linux" "i686-linux" "armhf-linux"
                       "aarch64-linux" "riscv64-linux")
                     #:configuration-file kernel-config))
```

Quaisquer chaves às quais não sejam atribuídos valores herdam seu valor padrão da definição `make-linux-libre`. Ao comparar os dois trechos acima, observe o comentário do código que se refere a `#:configuration-file`. Por causa disso, não é realmente fácil incluir uma configuração de kernel personalizada na definição, mas não se preocupe, existem outras maneiras de trabalhar com o que temos.

Existem duas maneiras de criar um kernel com uma configuração de kernel personalizada. A primeira é fornecer um arquivo `.config` padrão durante o processo de construção, incluindo um arquivo `.config` real como uma entrada nativa para nosso kernel personalizado. A seguir está um trecho da fase `'configure` personalizada da definição do pacote `make-linux-libre`:

```
(let ((build  (assoc-ref %standard-phases 'build))
      (config (assoc-ref (or native-inputs inputs) "kconfig")))
```

```
;; Use um arquivo de configuração de kernel personalizado ou um padrão
;; configuration file.
(if config
    (begin
      (copy-file config ".config")
      (chmod ".config" #o666))
    (invoke "make" ,defconfig)))
```

Abaixo está um exemplo de pacote de kernel. O pacote `linux-libre` não é nada especial e pode ser herdado e ter seus campos substituídos como qualquer outro pacote:

```
(define-public linux-libre/E2140
  (package
    (inherit linux-libre)
    (native-inputs
     `(("kconfig" ,(local-file "E2140.config"))
      ,@(alist-delete "kconfig"
                      (package-native-inputs linux-libre))))))
```

No mesmo diretório do arquivo que define `linux-libre-E2140` está um arquivo chamado `E2140.config`, que é um arquivo de configuração real do kernel. A palavra-chave `defconfig` de `make-linux-libre` é deixada em branco aqui, então a única configuração do kernel no pacote é aquela que foi incluída no campo `native-inputs`.

A segunda maneira de criar um kernel customizado é passar um novo valor para a palavra-chave `extra-options` do procedimento `make-linux-libre`. A palavra-chave `extra-options` funciona com outra função definida logo abaixo dela:

```
(define %default-extra-linux-options
  `(;; https://lists.gnu.org/archive/html/guix-devel/2014-04/msg00039.html
   ("CONFIG_DEVPTS_MULTIPLE_INSTANCES" . #true)
   ;; Modules required for initrd:
   ("CONFIG_NET_9P" . m)
   ("CONFIG_NET_9P_VIRTIO" . m)
   ("CONFIG_VIRTIO_BLK" . m)
   ("CONFIG_VIRTIO_NET" . m)
   ("CONFIG_VIRTIO_PCI" . m)
   ("CONFIG_VIRTIO_BALLOON" . m)
   ("CONFIG_VIRTIO_MMIO" . m)
   ("CONFIG_FUSE_FS" . m)
   ("CONFIG_CIFS" . m)
   ("CONFIG_9P_FS" . m)))

(define (config->string options)
  (string-join (map (match-lambda
                      ((option . 'm)
                       (string-append option "=m"))
                      ((option . #true)
                       (string-append option "=y"))
                      ((option . #false)
                       (string-append option "=n")))
```

```
                    options)
                  "\n"))
```

E no script de configuração personalizado do pacote ‘make-linux-libre‘:

```
;; A anexação funciona mesmo quando a opção não estava no arquivo.
;; A última prevalece se duplicada.
(let ((port (open-file ".config" "a"))
      (extra-configuration ,(config->string extra-options)))
  (display extra-configuration port)
  (close-port port))

(invoke "make" "oldconfig")
```

Portanto, ao não fornecer um arquivo de configuração, o `.config` começa em branco e então escrevemos nele a coleção de flags que desejamos. Aqui está outro kernel personalizado:

```
(define %macbook41-full-config
  (append %macbook41-config-options
          %file-systems
          %efi-support
          %emulation
          (@@ (gnu packages linux) %default-extra-linux-options)))

(define-public linux-libre-macbook41
  ;; XXX: Access the internal 'make-linux-libre*' procedure, which is
  ;; private and unexported, and is liable to change in the future.
  ((@@ (gnu packages linux) make-linux-libre*)
   (@@ (gnu packages linux) linux-libre-version)
   (@@ (gnu packages linux) linux-libre-gnu-revision)
   (@@ (gnu packages linux) linux-libre-source)
   '("x86_64-linux")
   #:extra-version "macbook41"
   #:extra-options %macbook41-config-options))
```

No exemplo acima, `%file-systems` é uma coleção de sinalizadores que habilitam suporte a diferentes sistemas de arquivos, `%efi-support` habilita suporte a EFI e `%emulation` habilita uma máquina x86_64-linux para atuar no modo de 32 bits também. O procedimento `default-extra-linux-options` é o definido acima, que teve que ser usado para evitar a perda das opções de configuração padrão da palavra-chave `extra-options`.

Tudo isso parece viável, mas como saber quais módulos são necessários para um sistema específico? Dois lugares que podem ser úteis para tentar responder a esta pergunta são o Gentoo Handbook (`https://wiki.gentoo.org/wiki/Handbook:AMD64/Installation/Kernel`) e o documentação do próprio kernel (`https://www .kernel.org/doc/html/latest/admin-guide/README.html?highlight=localmodconfig`). Pela documentação do kernel, parece que `make localmodconfig` é o comando que queremos.

Para realmente executar `make localmodconfig` primeiro precisamos obter e descompactar o código-fonte do kernel:

```
tar xf $(guix build linux-libre --source)
```

Uma vez dentro do diretório que contém o código-fonte, execute `touch .config` para criar um `.config` inicial e vazio para começar. `make localmodconfig` funciona vendo o que você já tem em `.config` e informando o que está faltando. Se o arquivo estiver em branco, você está perdendo tudo. O próximo passo é executar:

```
guix shell -D linux-libre -- make localmodconfig
```

e observe a saída. Observe que o arquivo `.config` ainda está vazio. A saída geralmente contém dois tipos de avisos. O primeiro começa com "WARNING" e pode ser ignorado no nosso caso. A segunda leitura:

```
module pcspkr did not have configs CONFIG_INPUT_PCSPKR
```

Para cada uma dessas linhas, copie a parte `CONFIG_XXXX_XXXX` para `.config` no diretório e anexe `=m`, para que no final fique assim:

```
CONFIG_INPUT_PCSPKR=m
CONFIG_VIRTIO=m
```

Após copiar todas as opções de configuração, execute `make localmodconfig` novamente para ter certeza de que você não tem nenhuma saída começando com "module". Depois de todos esses módulos específicos da máquina, restam mais alguns que também são necessários. `CONFIG_MODULES` é necessário para que você possa construir e carregar módulos separadamente e não ter tudo embutido no kernel. `CONFIG_BLK_DEV_SD` é necessário para leitura de discos rígidos. É possível que existam outros módulos dos quais você precisará.

Este post não pretende ser um guia para configurar seu próprio kernel, portanto, se você decidir construir um kernel personalizado, você terá que procurar outros guias para criar um kernel adequado às suas necessidades.

A segunda maneira de definir a configuração do kernel faz mais uso dos recursos do Guix e permite compartilhar segmentos de configuração entre diferentes kernels. Por exemplo, todas as máquinas que usam EFI para inicializar possuem vários sinalizadores de configuração EFI necessários. É provável que todos os kernels compartilhem uma lista de sistemas de arquivos para suporte. Ao usar variáveis, é mais fácil ver rapidamente quais recursos estão habilitados e garantir que você não tenha recursos em um kernel, mas ausentes em outro.

No entanto, não foi discutido o initrd do Guix e sua personalização. É provável que você precise modificar o initrd em uma máquina usando um kernel customizado, já que certos módulos que devem ser compilados podem não estar disponíveis para inclusão no initrd.

3.3 API de imagem do sistema Guix

Historicamente, o Sistema Guix é centrado em uma estrutura `operating-system`. Esta estrutura contém vários campos que vão desde o gerenciador de boot e a declaração do kernel até os serviços a serem instalados.

Dependendo da máquina de destino, que pode ir de uma máquina `x86_64` padrão a um pequeno computador de placa única ARM, como o Pine64, as restrições de imagem podem variar muito. Os fabricantes de hardware imporão diferentes formatos de imagem com vários tamanhos de partição e deslocamentos.

Para criar imagens adequadas para todas essas máquinas, é necessária uma nova abstração: esse é o objetivo do registro `image`. Este registro contém todas as informações

necessárias para ser transformada em uma imagem autônoma, que pode ser inicializada diretamente em qualquer máquina de destino.

```
(define-record-type* <image>
  image make-image
  image?
  (name               image-name ;symbol
                      (default #f))
  (format             image-format) ;symbol
  (target             image-target
                      (default #f))
  (size               image-size  ;size in bytes as integer
                      (default 'guess))
  (operating-system   image-operating-system  ;<operating-system>
                      (default #f))
  (partitions         image-partitions ;list of <partition>
                      (default '()))
  (compression?       image-compression? ;boolean
                      (default #t))
  (volatile-root?     image-volatile-root? ;boolean
                      (default #t))
  (substitutable?     image-substitutable? ;boolean
                      (default #t)))
```

Este registro contém o sistema operacional a ser instanciado. O campo `format` define o tipo de imagem e pode ser `efi-raw`, `qcow2` ou `iso9660` por exemplo. No futuro, poderá ser estendido para `docker` ou outros tipos de imagem.

Um novo diretório nas fontes do Guix é dedicado à definição de imagens. Por enquanto existem quatro arquivos:

- `gnu/system/images/hurd.scm`
- `gnu/system/images/pine64.scm`
- `gnu/system/images/novena.scm`
- `gnu/system/images/pinebook-pro.scm`

Vamos dar uma olhada em `pine64.scm`. Ele contém a variável `pine64-barebones-os` que é uma definição mínima de um sistema operacional dedicado à placa **Pine A64 LTS**.

```
(define pine64-barebones-os
  (operating-system
   (host-name "vignemale")
   (timezone "Europe/Paris")
   (locale "en_US.utf8")
   (bootloader (bootloader-configuration
                (bootloader u-boot-pine64-lts-bootloader)
                (targets '("/dev/vda"))))
   (initrd-modules '())
   (kernel linux-libre-arm64-generic)
   (file-systems (cons (file-system
                        (device (file-system-label "my-root"))
```

```
                        (mount-point "/")
                        (type "ext4"))
                       %base-file-systems))
    (services (cons (service agetty-service-type
                             (agetty-configuration
                              (extra-options '("-L")) ; sem detecção de portadora
                              (baud-rate "115200")
                              (term "vt100")
                              (tty "ttyS0")))
                    %base-services))))
```

Os campos `kernel` e `bootloader` apontam para pacotes dedicados a esta placa.

Logo abaixo, a variável `pine64-image-type` também está definida.

```
(define pine64-image-type
  (image-type
   (name 'pine64-raw)
   (constructor (cut image-with-os arm64-disk-image <>))))
```

Ele está usando um registro do qual ainda não falamos, o registro `image-type`, definido desta forma:

```
(define-record-type* <image-type>
  image-type make-image-type
  image-type?
  (name           image-type-name) ;symbol
  (constructor    image-type-constructor)) ;<operating-system> -> <image>
```

O objetivo principal deste registro é associar um nome a um procedimento que transforma um isso é necessário, vamos dar uma olhada no comando que produz uma imagem de um arquivo de configuração `operating-system`:

```
guix system image my-os.scm
```

Este comando espera uma configuração `operating-system` mas como devemos indicar que queremos uma imagem direcionada a uma placa Pine64? Precisamos fornecer uma informação extra, o `image-type`, passando o sinalizador `--image-type` ou `-t`, desta forma:

```
guix system image --image-type=pine64-raw my-os.scm
```

Este parâmetro `image-type` aponta para o `pine64-image-type` definido acima. Portanto, ao `operating-system` declarado em `my-os.scm` será aplicado o procedimento `(cut image-with-os arm64-disk-image <>)` para transformá-lo em um imagem.

A imagem resultante se parece com:

```
(image
 (format 'disk-image)
 (target "aarch64-linux-gnu")
 (operating-system my-os)
 (partitions
  (list (partition
         (inherit root-partition)
         (offset root-offset)))))
```

que é a agregação do `operating-system` definido em `my-os.scm` ao registro `arm64-disk-image`.

Mas chega de loucura do Scheme. O que essa API de imagem traz para o usuário do Guix?

Pode-se executar:

```
mathieu@cervin:~$ guix system --list-image-types
Os tipos de imagem disponíveis são:

   - unmatched-raw
   - rock64-raw
   - pinebook-pro-raw
   - pine64-raw
   - novena-raw
   - hurd-raw
   - hurd-qcow2
   - qcow2
   - iso9660
   - uncompressed-iso9660
   - tarball
   - efi-raw
   - mbr-raw
   - docker
   - wsl2
   - raw-with-offset
   - efi32-raw
```

e escrevendo um arquivo `operating-system` baseado em `pine64-barebones-os`, você pode personalizar sua imagem de acordo com suas preferências em um arquivo (`my-pine-os.scm`) como este :

```
(use-modules (gnu services linux)
             (gnu system images pine64))

(let ((base-os pine64-barebones-os))
  (operating-system
    (inherit base-os)
    (timezone "America/Indiana/Indianapolis")
    (services
     (cons
      (service earlyoom-service-type
               (earlyoom-configuration
                (prefer-regexp "icecat|chromium")))
      (operating-system-user-services base-os)))))
```

execute:

```
guix system image --image-type=pine64-raw my-pine-os.scm
```

ou,

```
guix system image --image-type=hurd-raw my-hurd-os.scm
```

para obter uma imagem que pode ser gravada diretamente em um disco rígido e inicializada.

Sem alterar nada em `my-hurd-os.scm`, chamando:

```
guix system image --image-type=hurd-qcow2 my-hurd-os.scm
```

em vez disso, produzirá uma imagem Hurd QEMU.

3.4 Usando chaves de segurança

O uso de chaves de segurança pode melhorar sua segurança, fornecendo uma segunda fonte de autenticação que não pode ser facilmente roubada ou copiada, pelo menos para um adversário remoto (algo que você possui), para o segredo principal (uma senha - algo que você conhece). , reduzindo o risco de falsificação de identidade.

O exemplo de configuração detalhado abaixo mostra qual configuração mínima precisa ser feita em seu sistema Guix para permitir o uso de uma chave de segurança Yubico. Espera-se que a configuração também possa ser útil para outras chaves de segurança, com pequenos ajustes.

3.4.1 Configuração para uso como autenticador de dois fatores (2FA)

Para serem utilizáveis, as regras do udev do sistema devem ser estendidas com regras específicas de chave. O seguinte mostra como estender suas regras do udev com o arquivo de regras do udev `lib/udev/rules.d/70-u2f.rules` fornecido pelo pacote `libfido2` do `(gnu packages security- token)` e adicione seu usuário ao grupo ‘`"plugdev"`’ que ele usa:

```
(use-package-modules ... security-token ...)
...
(operating-system
 ...
 (users (cons* (user-account
                (name "your-user")
                (group "users")
                (supplementary-groups
'("wheel" "netdev" "audio" "video"
                   "plugdev"))           ;<- grupo de sistema adicionado
                (home-directory "/home/your-user"))
               %base-user-accounts))
 ...
 (services
  (cons*
   ...
   (udev-rules-service 'fido2 libfido2 #:groups '("plugdev")))))
```

Depois de reconfigurar seu sistema e fazer login novamente em sua sessão gráfica para que o novo grupo esteja em vigor para seu usuário, você pode verificar se sua chave pode ser usada iniciando:

```
guix shell ungoogled-chromium -- chromium chrome://settings/securityKeys
```

e validar que a chave de segurança pode ser redefinida através do menu "Redefinir sua chave de segurança". Se funcionar, parabéns, sua chave de segurança está pronta para ser usada com aplicativos que suportam autenticação de dois fatores (2FA).

3.4.2 Desativando a geração de código OTP para um Yubikey

Se você usa uma chave de segurança Yubikey e fica irritado com os códigos OTP falsos que ela gera ao tocar inadvertidamente na chave (por exemplo, fazendo com que você se torne um spammer no canal '`#guix`' ao discutir sobre seu cliente de IRC favorito!), você pode desativá-lo através do seguinte comando `ykman`:

```
guix shell python-yubikey-manager -- ykman config usb --force --disable OTP
```

Alternativamente, você pode usar o comando `ykman-gui` fornecido pelo pacote `yubikey-manager-qt` e desativar totalmente o aplicativo '`OTP`' para a interface USB ou, a partir do pacote '`Applications -> Visualização OTP`', exclua a configuração do slot 1, que vem pré-configurada com o aplicativo Yubico OTP.

3.4.3 Exigindo que um Yubikey abra um banco de dados KeePassXC

O aplicativo gerenciador de senhas KeePassXC tem suporte para Yubikeys, mas requer a instalação de regras udev para seu sistema Guix e algumas configurações do aplicativo Yubico OTP na chave.

O arquivo de regras do udev necessário vem do pacote `yubikey-personalization` e pode ser instalado como:

```
(use-package-modules ... security-token ...)
...
(operating-system
 ...
 (services
  (cons*
   ...
   (udev-rules-service 'yubikey yubikey-personalization))))
```

Depois de reconfigurar seu sistema (e reconectar seu Yubikey), você desejará configurar o aplicativo de desafio/resposta OTP de seu Yubikey em seu slot 2, que é o que o KeePassXC usa. É fácil fazer isso por meio da ferramenta de configuração gráfica Yubikey Manager, que pode ser invocada com:

```
guix shell yubikey-manager-qt -- ykman-gui
```

Primeiro, certifique-se de que '`OTP`' esteja habilitado na aba '`Interfaces`', depois navegue até '`Applications -> OTP`' e clique no botão '`Configure`' abaixo de '`Long Touch (Slot 2)`' seção. Selecione '`Challenge-response`', insira ou gere uma chave secreta e clique no botão '`Finish`'. Se você tiver um segundo Yubikey que gostaria de usar como backup, você deve configurá-lo da mesma forma, usando a chave secreta *mesma*.

Seu Yubikey agora deve ser detectado pelo KeePassXC. Ele pode ser adicionado a um banco de dados navegando até o menu '`Database -> Database Security...`' do KeePassXC e clicando no botão '`Adicionar proteção adicional...`' e em '`Add Challenge-Response`', selecionando a chave de segurança no menu suspenso e clicando no botão '`OK`' para concluir a configuração.

3.5 Trabalho mcron de DNS dinâmico

Se o seu ISP (Internet Service Provider) fornece apenas endereços IP dinâmicos, pode ser útil configurar um serviço DNS (Domain Name System) dinâmico (também conhecido como DDNS (Dynamic DNS)) para associar um nome de host estático para um endereço IP público, mas dinâmico. Existem vários serviços que podem ser usados para isso; no job mcron a seguir, DuckDNS (`https://duckdns.org`) é usado. Também deve funcionar com outros serviços DNS dinâmicos que oferecem uma interface semelhante para atualizar o endereço IP.

O Job mcron é fornecido abaixo, onde *DOMAIN* deve ser substituído pelo seu próprio prefixo de domínio, e o token fornecido pelo DuckDNS é associado a *DOMAIN* e adicionado ao arquivo `/etc/duckdns/`*DOMAIN* `arquivo .token`.

```
(define duckdns-job
  ;; Atualize o IP do domínio pessoal a cada 5 minutos.
  #~(job '(next-minute (range 0 60 5))
 #$(program-file
            "duckdns-update"
            (with-extensions (list guile-gnutls) ;required by (web client)
              #~(begin
                   (use-modules (ice-9 textual-ports)
                                (web client))
                   (let ((token (string-trim-both
                                 (call-with-input-file "/etc/duckdns/DOMAIN.token"
                                   get-string-all)))
                         (query-template (string-append "https://www.duckdns.org/"
                                                        "update?domains=DOMAIN"
                                                        "&token=~a&ip=")))
                     (http-get (format #f query-template token))))))
          "duckdns-update"
          #:user "nobody"))
```

O Job então precisa ser adicionado à lista de trabalhos mcron do seu sistema, usando algo como:

```
(operating-system
 (services
  (cons* (service mcron-service-type
           (mcron-configuration
             (jobs (list duckdns-job ...))))
         ...
         %base-services)))
```

3.6 Conectando-se à VPN Wireguard

Para se conectar a um servidor VPN Wireguard, você precisa que o módulo do kernel esteja carregado na memória e um pacote que forneça ferramentas de rede que o suportem (por exemplo, `wireguard-tools` ou `network-manager`).

Aqui está um exemplo de configuração para Linux-Libre versão menor que 5.6, onde o módulo está fora da árvore e precisa ser carregado manualmente — as seguintes revisões do kernel o possuem integrado e, portanto, não precisam de tal configuração:

```
(use-modules (gnu))
(use-service-modules desktop)
(use-package-modules vpn)

(operating-system
  ;; ...
  (services (cons (simple-service 'wireguard-module
                                  kernel-module-loader-service-type
                                  '("wireguard"))
                  %desktop-services))
  (packages (cons wireguard-tools %base-packages))
  (kernel-loadable-modules (list wireguard-linux-compat)))
```

Após reconfigurar e reiniciar seu sistema, você pode usar as ferramentas Wireguard ou NetworkManager para conectar-se a um servidor VPN.

3.6.1 Usando ferramentas Wireguard

Para testar a configuração do Wireguard é conveniente usar `wg-quick`. Basta fornecer um arquivo de configuração `wg-quick up ./wg0.conf`; ou coloque esse arquivo em `/etc/wireguard` e execute `wg-quick up wg0`.

> **Nota:** Esteja avisado que o autor descreveu este comando como um: "[. . .] script bash muito rápido e sujo [. . .]".

3.6.2 Usando o NetworkManager

Graças ao suporte do NetworkManager para Wireguard, podemos conectar-nos à nossa VPN usando o comando `nmcli`. Até este ponto, este guia pressupõe que você esteja usando o serviço Network Manager fornecido por `%desktop-services`. Caso contrário, você precisará ajustar sua lista de serviços para carregar `network-manager-service-type` e reconfigurar seu sistema Guix.

Para importar sua configuração VPN, execute o comando "nmcli import":

```
# nmcli connection import type wireguard file wg0.conf
Connection 'wg0' (edbee261-aa5a-42db-b032-6c7757c60fde) successfully added
```

Isso criará um arquivo de configuração em `/etc/NetworkManager/wg0.nmconnection`. Em seguida, conecte-se ao servidor Wireguard:

```
$ nmcli connection up wg0
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/
```

Por padrão, o NetworkManager se conectará automaticamente na inicialização do sistema. Para mudar esse comportamento você precisa editar sua configuração:

```
# nmcli connection modify wg0 connection.autoconnect no
```

Para obter informações mais específicas sobre NetworkManager e wireguard consulte este artigo (`https://blogs.gnome.org/thaller/2019/03/15/wireguard-in-networkmanager/`).

3.7 Customizando um Gerenciador de Janelas

3.7.1 StumpWM

Você pode instalar o StumpWM com um sistema Guix adicionando pacotes `stumpwm` e opcionalmente `` `(,stumpwm "lib") `` a um arquivo de configuração do sistema, por exemplo, `/etc/config.scm`.

Um exemplo de configuração pode ser assim:

```
(use-modules (gnu))
(use-package-modules wm)

(operating-system
  ;; ...
  (packages (append (list sbcl stumpwm `(,stumpwm "lib"))
                    %base-packages)))
```

Por padrão, o StumpWM usa fontes X11, que podem ser pequenas ou pixeladas em seu sistema. Você pode corrigir isso instalando o módulo StumpWM contrib Lisp `sbcl-ttf-fonts`, adicionando-o aos pacotes do sistema Guix:

```
(use-modules (gnu))
(use-package-modules fonts wm)

(operating-system
  ;; ...
  (packages (append (list sbcl stumpwm `(,stumpwm "lib"))
                    sbcl-ttf-fonts font-dejavu %base-packages)))
```

Então você precisa adicionar o seguinte código a um arquivo de configuração do StumpWM `~/.stumpwm.d/init.lisp`:

```
(require :ttf-fonts)
(setf xft:*font-dirs* '("/run/current-system/profile/share/fonts/"))
(setf clx-truetype:+font-cache-filename+ (concat (getenv "HOME")
                                                 "/.fonts/font-cache.sexp"))
(xft:cache-fonts)
(set-font (make-instance 'xft:font :family "DejaVu Sans Mono"
                                   :subfamily "Book" :size 11))
```

3.7.2 Bloqueio de sessão

Dependendo do seu ambiente, o bloqueio da tela da sua sessão pode ser incorporado ou pode ser algo que você mesmo precisa configurar. Se você usa um ambiente de área de trabalho como GNOME ou KDE, ele geralmente está integrado. Se você usa um gerenciador de janelas simples como StumpWM ou EXWM, talvez seja necessário configurá-lo você mesmo.

3.7.2.1 Xorg

Se você usa Xorg, você pode usar o utilitário xss-lock (`https://www.mankier.com/1/xss-lock`) para bloquear a tela da sua sessão. xss-lock é acionado pelo DPMS que, desde o Xorg 1.8, é detectado automaticamente e habilitado se o ACPI também estiver habilitado no tempo de execução do kernel.

Para usar o xss-lock, você pode simplesmente executá-lo e colocá-lo em segundo plano antes de iniciar o gerenciador de janelas, por exemplo, seu `~/.xsession`:

```
xss-lock -- slock &
exec stumpwm
```

Neste exemplo, xss-lock usa `slock` para fazer o bloqueio real da tela quando determina que é apropriado, como quando você suspende seu dispositivo.

Para que o slock possa ser um bloqueador de tela para a sessão gráfica, ele precisa ser definido como setuid-root para poder autenticar usuários e precisa de um serviço PAM. Isso pode ser conseguido adicionando o seguinte serviço ao seu `config.scm`:

```
(service screen-locker-service-type
         (screen-locker-configuration
          (name "slock")
          (program (file-append slock "/bin/slock"))))
```

Se você bloquear sua tela manualmente, por exemplo, chamando slock diretamente quando quiser bloquear sua tela, mas não suspendê-la, é uma boa ideia notificar xss-lock sobre isso para que não ocorra confusão. Isso pode ser feito executando `xset s activate` imediatamente antes de executar o slock.

3.8 Executando Guix em um Servidor Linode

Para executar o Guix num servidor hospedado por Linode (`https://www.linode.com`), comece com um servidor Debian recomendado. Recomendamos usar a distribuição padrão como forma de inicializar o Guix. Crie suas chaves SSH.

```
ssh-keygen
```

Certifique-se de adicionar sua chave SSH para facilitar o login no servidor remoto. Isto é feito trivialmente através da interface gráfica do Linode para adicionar chaves SSH. Vá para o seu perfil e clique em adicionar chave SSH. Copie nele a saída de:

```
cat ~/.ssh/<username>_rsa.pub
```

Desligue o Linode.

Na aba Armazenamento do Linode, redimensione o disco Debian para ser menor. Recomenda-se 30 GB de espaço livre. Em seguida, clique em "Adicionar um disco" e preencha o formulário com o seguinte:

- Label: "Guix"
- Filesystem: ext4
- Defina-o para o tamanho restante

Na aba Configurações, pressione “Editar” no perfil Debian padrão. Em "Bloquear atribuição de dispositivo", clique em "Adicionar um dispositivo". Deve ser `/dev/sdc` e você pode selecionar o disco "Guix". Salvar alterações.

Agora "Adicionar uma configuração", com o seguinte:

- Etiqueta: Guix
- Kernel:GRUB 2 (está na parte inferior! Esta etapa é **IMPORTANTE!**)
- Bloquear atribuição de dispositivo:
- `/dev/sda`: Guix

- `/dev/sdb`: swap
- Dispositivo raiz: `/dev/sda`
- Desligue todos os auxiliares de sistema de arquivos/inicialização

Agora ligue-o novamente, inicializando com a configuração do Debian. Quando estiver em execução, faça ssh para o seu servidor via `ssh root@<your-server-IP-here>`. (Você pode encontrar o endereço IP do seu servidor na seção Resumo do Linode.) Agora você pode executar as etapas "instalar o guix de veja Seção “Instalação de binários” em *GNU Guix*":

```
sudo apt-get install gpg
wget https://sv.gnu.org/people/viewgpg.php?user_id=15145 -qO - | gpg --import -
wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
chmod +x guix-install.sh
./guix-install.sh
guix pull
```

Agora é hora de escrever uma configuração para o servidor. As principais informações estão abaixo. Salve o arquivo resultante como `guix-config.scm`.

```
(use-modules (gnu)
             (guix modules))
(use-service-modules networking
                     ssh)
(use-package-modules admin
                     package-management
                     ssh
                     tls)

(operating-system
  (host-name "my-server")
  (timezone "America/New_York")
  (locale "en_US.UTF-8")
  ;; Este código bobo irá gerar o grub.cfg
  ;; sem instalar o bootloader grub no disco.
  (bootloader (bootloader-configuration
               (bootloader
                (bootloader
                 (inherit grub-bootloader)
                 (installer #~(const #true))))))
  (file-systems (cons (file-system
                        (device "/dev/sda")
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))


  (swap-devices (list "/dev/sdb"))
```

```
(initrd-modules (cons "virtio_scsi" ; Necessário para encontrar o disco
                      %base-initrd-modules))

(users (cons (user-account
              (name "janedoe")
              (group "users")
              ;; Adicionando a conta ao grupo "wheel"
              ;; o torna um super-usuário
              (supplementary-groups '("wheel"))
              (home-directory "/home/janedoe"))
             %base-user-accounts))

(packages (cons* openssh-sans-x
                 %base-packages))

(services (cons*
           (service dhcp-client-service-type)
           (service openssh-service-type
                    (openssh-configuration
                     (openssh openssh-sans-x)
                     (password-authentication? #false)
                     (authorized-keys
                      `(("janedoe" ,(local-file "janedoe_rsa.pub"))
                        ("root" ,(local-file "janedoe_rsa.pub"))))))
           %base-services)))
```

Substitua os seguintes campos na configuração acima:

```
(host-name "my-server"); substitua pelo nome do seu servidor
; se você escolheu um servidor linode fora dos EUA, então
; use tzselect para encontrar uma string de fuso horário correta
(timezone "America/New_York"); se necessário, substitua o fuso horário
(name "janedoe"); substitua pelo seu nome de usuário
("janedoe" ,(local-file "janedoe_rsa.pub")) ; substitua pela sua chave ssh
("root" ,(local-file "janedoe_rsa.pub")) ; substitua pela sua chave ssh
```

A última linha no exemplo acima permite que você faça login no servidor como root e defina a senha root inicial (veja a nota no final desta receita sobre login root). Depois de fazer isso, você pode excluir essa linha da sua configuração e reconfigurar para evitar o login root.

Copie sua chave pública ssh (por exemplo: `~/.ssh/id_rsa.pub`) como `<seu-nome-de-usuário-aqui>_rsa.pub` e coloque `guix-config.scm` no mesmo diretório. Em um novo terminal execute estes comandos.

```
sftp root@<endereço IP do servidor remoto>
put /caminho/para/arquivos/<nome de usuário>_rsa.pub .
put /caminho/para/arquivos/guix-config.scm .
```

No seu primeiro terminal, monte o drive guix:

```
mkdir /mnt/guix
mount /dev/sdc /mnt/guix
```

Devido à forma como configuramos a seção bootloader do guix-config.scm, apenas o arquivo de configuração grub será instalado. Então, precisamos copiar algumas das outras coisas do GRUB já instaladas no sistema Debian:

```
mkdir -p /mnt/guix/boot/grub
cp -r /boot/grub/* /mnt/guix/boot/grub/
```

Agora inicialize a instalação do Guix:

```
guix system init guix-config.scm /mnt/guix
```

Ok, desligue-o! Agora, no console Linode, selecione boot e selecione "Guix".

Depois de inicializar, você poderá fazer login via SSH! (A configuração do servidor terá mudado.) Você pode encontrar um erro como:

```
$ ssh root@<server ip address>
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:0B+wp33w57AnKQuHCvQP0+ZdKaqYrI/kyU7CfVbS7R4.
Please contact your system administrator.
Add correct host key in /home/joshua/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /home/joshua/.ssh/known_hosts:3
ECDSA host key for 198.58.98.76 has changed and you have requested strict checking.
Host key verification failed.
```

Exclua o arquivo `~/.ssh/known_hosts` ou exclua a linha incorreta começando com o endereço IP do seu servidor.

Certifique-se de definir sua senha e a senha do root.

```
ssh root@<endereço IP remoto>
passwd; para a senha root
passwd <nome de usuário>; para a senha do usuário
```

Talvez você não consiga executar os comandos acima neste momento. Se você tiver problemas para fazer login remotamente em sua caixa linode via SSH, então você ainda pode precisar definir sua senha root e de usuário inicialmente clicando na opção "Launch Console" em seu linode. Escolha "Glish" em vez de "Weblish". Agora você deve conseguir fazer o ssh na máquina.

Viva! Neste ponto você pode desligar o servidor, excluir o disco Debian e redimensionar o Guix para o restante do tamanho. Parabéns!

A propósito, se você salvá-lo como uma imagem de disco neste momento, será fácil criar novas imagens Guix! Pode ser necessário reduzir o tamanho da imagem Guix para 6144 MB para salvá-la como uma imagem. Então você pode redimensioná-lo novamente para o tamanho máximo.

3.9 Executando Guix em um servidor Kimsufi

Para executar o Guix em um servidor hospedado por Kimsufi (`https://www.kimsufi.com/`), clique na guia netboot, selecione Rescue64-pro e reinicie.

A OVH enviar-lhe-á por e-mail as credenciais necessárias para efetuar o ssh num sistema Debian.

Agora você pode executar as etapas "instalar guix de veja Seção “Instalação de binários” em *GNU Guix*":

```
wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
chmod +x guix-install.sh
./guix-install.sh
guix pull
```

Particione as unidades e formate-as, primeiro interrompa a matriz raid:

```
mdadm --stop /dev/md127
mdadm --zero-superblock /dev/sda2 /dev/sdb2
```

Em seguida, limpe os discos e configure as partições, criaremos uma matriz RAID 1.

```
wipefs -a /dev/sda
wipefs -a /dev/sdb

parted /dev/sda --align=opt -s -m -- mklabel gpt
parted /dev/sda --align=opt -s -m -- \
 mkpart bios_grub 1049kb 512MiB \
 set 1 bios_grub on
parted /dev/sda --align=opt -s -m -- \
 mkpart primary 512MiB -512MiB
 set 2 raid on
parted /dev/sda --align=opt -s -m -- mkpart primary linux-swap 512MiB 100%

parted /dev/sdb --align=opt -s -m -- mklabel gpt
parted /dev/sdb --align=opt -s -m -- \
     mkpart bios_grub 1049kb 512MiB \
     set 1 bios_grub on
parted /dev/sdb --align=opt -s -m -- \
     mkpart primary 512MiB -512MiB \
     set 2 raid on
parted /dev/sdb --align=opt -s -m -- mkpart primary linux-swap 512MiB 100%
```

Crie a matriz:

```
mdadm --create /dev/md127 --level=1 --raid-disks=2 \
  --metadata=0.90 /dev/sda2 /dev/sdb2
```

Agora crie sistemas de arquivos nas partições relevantes, primeiro as partições de inicialização:

```
mkfs.ext4  /dev/sda1
mkfs.ext4  /dev/sdb1
```

Então a partição raiz:

```
mkfs.ext4 /dev/md127
```

Inicialize as partições swap:

```
mkswap /dev/sda3
swapon /dev/sda3
mkswap /dev/sdb3
swapon /dev/sdb3
```

Monte a unidade guix:

```
mkdir /mnt/guix
mount /dev/md127 /mnt/guix
```

Agora é hora de escrever um arquivo `os.scm` de declaração do sistema operacional; aqui está uma amostra:

```
(use-modules (gnu) (guix))
(use-service-modules networking ssh vpn virtualization sysctl admin mcron)
(use-package-modules ssh tls tmux vpn virtualization)

(operating-system
  (host-name "kimsufi")

  (bootloader (bootloader-configuration
       (bootloader grub-bootloader)
       (targets (list "/dev/sda" "/dev/sdb"))
       (terminal-outputs '(console))))

  ;; Adicionar um módulo de kernel para RAID-1 (conhecido como. "espelho").
  (initrd-modules (cons* "raid1"  %base-initrd-modules))

  (mapped-devices
   (list (mapped-device
          (source (list "/dev/sda2" "/dev/sdb2"))
          (target "/dev/md127")
          (type raid-device-mapping))))

  (swap-devices
   (list (swap-space
          (target "/dev/sda3"))
         (swap-space
          (target "/dev/sdb3"))))

  (issue
   ;;Conteúdo padrão para /etc/issue.
   "\
Este é o sistema GNU em Kimsufi. Bem-vindo.\n")

  (file-systems (cons* (file-system
         (mount-point "/")
         (device "/dev/md127")
         (type "ext4")
```

```
          (dependencies mapped-devices))
        %base-file-systems))

  (users (cons (user-account
        (name "guix")
        (comment "guix")
        (group "users")
        (supplementary-groups '("wheel"))
        (home-directory "/home/guix"))
       %base-user-accounts))

  (sudoers-file
   (plain-file "sudoers" "\
root ALL=(ALL) ALL
%wheel ALL=(ALL) ALL
guix ALL=(ALL) NOPASSWD:ALL\n"))

  ;; Pacotes instalados globalmente.
  (packages (cons* tmux nss-certs gnutls wireguard-tools %base-packages))
  (services
   (cons*
    (service static-networking-service-type
     (list (static-networking
    (addresses (list (network-address
      (device "enp3s0")
      (value "server-ip-address/24"))))
    (routes (list (network-route
   (destination "default")
   (gateway "server-gateway"))))
    (name-servers '("213.186.33.99")))))

    (service unattended-upgrade-service-type)

    (service openssh-service-type
     (openssh-configuration
      (openssh openssh-sans-x)
      (permit-root-login #f)
      (authorized-keys
       `(("guix" ,(plain-file "ssh-key-name.pub"
                              "ssh-public-key-content"))))))
    (modify-services %base-services
      (sysctl-service-type
       config =>
       (sysctl-configuration
(settings (append '(("net.ipv6.conf.all.autoconf" . "0")
    ("net.ipv6.conf.all.accept_ra" . "0"))
  %default-sysctl-settings)))))))
```

Não se esqueça de substituir as variáveis *server-ip-address*, *server-gateway*, *ssh-key-name* e *ssh-public-key-content* pelos seus próprias valores.

O gateway é o último IP utilizável em seu bloco, portanto, se você tiver um servidor com IP '`37.187.79.10`', seu gateway será '`37.187.79.254`'.

Transfira o arquivo de declaração do sistema operacional `os.scm` para o servidor por meio dos comandos `scp` ou `sftp`.

Agora só falta instalar o Guix com `guix system init` e reiniciar.

No entanto, primeiro precisamos configurar um chroot, porque a partição raiz do sistema de recuperação é montada em uma partição aufs e se você tentar instalar o Guix ele falhará na etapa de instalação do GRUB reclamando do caminho canônico de "aufs".

Instale os pacotes que serão usados no chroot:

```
guix install bash-static parted util-linux-with-udev coreutils guix
```

Em seguida, execute o seguinte para criar os diretórios necessários para o chroot:

```
cd /mnt && \
mkdir -p bin etc gnu/store root/.guix-profile/ root/.config/guix/current \
  var/guix proc sys dev
```

Copie o host resolv.conf no chroot:

```
cp /etc/resolv.conf etc/
```

Monte os dispositivos de bloco, o armazém e seu banco de dados e a configuração atual do guix:

```
mount --rbind /proc /mnt/proc
mount --rbind /sys /mnt/sys
mount --rbind /dev /mnt/dev
mount --rbind /var/guix/ var/guix/
mount --rbind /gnu/store gnu/store/
mount --rbind /root/.config/ root/.config/
mount --rbind /root/.guix-profile/bin/ bin
mount --rbind /root/.guix-profile root/.guix-profile/
```

Faça chroot em /mnt e instale o sistema:

```
chroot /mnt/ /bin/bash

guix system init /root/os.scm /guix
```

Por fim, na interface do usuário (IU) da web, altere '`netboot`' para '`boot to disk`' e reinicie (também na IU da web).

Aguarde alguns minutos e tente fazer ssh com `ssh guix@`*`endereço IP do servidor`*`> -i `*`caminho para sua chave ssh`*

Você deve ter um sistema Guix instalado e funcionando no Kimsufi; Parabéns!

3.10 Configurando uma montagem vinculada

Para vincular a montagem de um sistema de arquivos, é necessário primeiro configurar algumas definições antes da seção `operating-system` da definição do sistema. Neste exemplo, vincularemos a montagem de uma pasta de uma unidade de disco rígido a `/tmp`, para evitar desgaste no SSD primário, sem dedicar uma partição inteira para ser montada como `/tmp`.

Primeiro, a unidade de origem que hospeda a pasta que desejamos vincular a montagem deve ser definida, para que a montagem de ligação possa depender dela.

```
(define source-drive ;; "source-drive" pode ser nomeado como você quiser.
   (file-system
    (device (uuid "UUID vai aqui"))
    (mount-point "/path-to-spinning-disk-goes-here")
    (type "ext4"))) ;; Certifique-se de definir isso para o tipo apropriado para sua u
```

A pasta de origem também deve ser definida, para que o guix saiba que não é um dispositivo de bloco normal, mas uma pasta.

```
;; "%source-directory" pode receber qualquer nome de variável válido.
(define (%source-directory) "/caminho_para_ o_disco_vai_aqui/tmp")
```

Finalmente, dentro da definição `file-systems`, devemos adicionar a própria montagem.

```
(file-systems (cons*

               ...<other drives omitted for clarity>...

               ;; Deve corresponder ao nome que você deu à unidade
               ;; de origem na definição anterior.
               source-drive

               (file-system
                ;; Certifique-se de que "source-directory" corresponde
                ;; à sua definição anterior.
                (device (%source-directory))
                (mount-point "/tmp")
                ;; Estamos montando uma pasta, não uma partição, então
                ;; esse tipo precisa ser "none"
                (type "none")
                (flags '(bind-mount))
                ;; Certifique-se de que "source-drive" corresponde ao
                ;; nome que você deu à variável para a unidade
                (dependencies (list source-drive))
                )

                ...<outras unidades omitidas para maior clareza>...

               ))
```

3.11 Obtendo substitutos pelo Tor

O daemon Guix pode usar um proxy HTTP para obter substitutos, aqui estamos configurando-o para obtê-los via Tor.

> **Aviso:** *Nem todo* o tráfego do daemon Guix passará pelo Tor! Somente HTTP/HTTPS será encaminhado ao proxy; As conexões FTP, o protocolo Git, SSH, etc. ainda passarão pela rede aberta. Novamente, esta configuração

não é infalível, pois parte do seu tráfego não será roteado pelo Tor. Use-o por sua conta e risco.

Observe também que o procedimento descrito aqui se aplica apenas à substituição de pacotes. Ao atualizar sua distribuição guix com `guix pull`, você ainda precisará usar `torsocks` se quiser rotear a conexão para os servidores de repositório git do guix através do Tor.

O servidor substituto do Guix está disponível como um serviço Onion, se você quiser usá-lo para obter seus substitutos através do Tor, configure seu sistema da seguinte forma:

```
(use-modules (gnu))
(use-service-module base networking)

(operating-system
  ...
  (services
    (cons
      (service tor-service-type
               (tor-configuration
                 (config-file (plain-file "tor-config"
                                          "HTTPTunnelPort 127.0.0.1:9250"))))
      (modify-services %base-services
        (guix-service-type
          config => (guix-configuration
                      (inherit config)
                      ;; ci.guix.gnu.org's  Serviço onion
                      (substitute-urls
                       "https://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclc
                      (http-proxy "http://localhost:9250")))))))
```

Isso manterá um processo tor em execução que fornece um túnel HTTP CONNECT que será usado por `guix-daemon`. O daemon pode usar outros protocolos além do HTTP(S) para obter recursos remotos. A solicitação usando esses protocolos não passará pelo Tor, pois estamos apenas configurando um túnel HTTP aqui. Observe que `substitutes-urls` está usando HTTPS e não HTTP ou não funcionará, isso é uma limitação do túnel do Tor; você pode querer usar `privoxy` para evitar tais limitações.

Se você não deseja sempre obter substitutos através do Tor, mas usá-lo apenas algumas vezes, pule o `guix-configuration`. Quando você deseja obter um substituto da execução do túnel Tor:

```
sudo herd set-http-proxy guix-daemon http://localhost:9250
guix build \
  --substitute-urls=https://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclcyd.o
```

3.12 Configurando NGINX com Lua

O NGINX pode ser estendido com scripts Lua.

Guix fornece serviço NGINX com capacidade de carregar módulos Lua e pacotes Lua específicos, e responder a solicitações avaliando scripts Lua.

O exemplo a seguir demonstra a definição do sistema com configuração para avaliar o script Lua `index.lua` na solicitação HTTP para o endpoint `http://localhost/hello`:

```
local shell = require "resty.shell"

local stdin = ""
local timeout = 1000  -- ms
local max_size = 4096  -- byte

local ok, stdout, stderr, reason, status =
   shell.run([[/run/current-system/profile/bin/ls /tmp]], stdin, timeout, max_size)

ngx.say(stdout)
```

```
(use-modules (gnu))
(use-service-modules #;... web)
(use-package-modules #;... lua)
(operating-system
  ;; ...
  (services
   ;; ...
   (service nginx-service-type
            (nginx-configuration
             (modules
              (list
               (file-append nginx-lua-module "/etc/nginx/modules/ngx_http_lua_module.s
             (lua-package-path (list lua-resty-core
                                     lua-resty-lrucache
                                     lua-resty-signal
                                     lua-tablepool
                                     lua-resty-shell))
             (lua-package-cpath (list lua-resty-signal))
             (server-blocks
              (list (nginx-server-configuration
                     (server-name '("localhost"))
                     (listen '("80"))
                     (root "/etc")
                     (locations (list
                                 (nginx-location-configuration
                                  (uri "/hello")
                                  (body (list #~(format #f "content_by_lua_file ~s;"
                                                        #$(local-file "index.lua")))))
```

3.13 Servidor de música com áudio Bluetooth

MPD, o Music Player Daemon, é um aplicativo flexível do lado do servidor para tocar música. Programas clientes em diferentes máquinas na rede — um telefone celular, um laptop, uma estação de trabalho de mesa — podem se conectar a ele para controlar a

reprodução de arquivos de áudio da sua coleção de música local. O MPD decodifica os arquivos de áudio e os reproduz em uma ou muitas saídas.

Por padrão, o MPD reproduzirá no dispositivo de áudio padrão. No exemplo abaixo, tornamos as coisas um pouco mais interessantes configurando um servidor de música headless. Não haverá interface gráfica de usuário, nenhum daemon Pulseaudio e nenhuma saída de áudio local. Em vez disso, configuraremos o MPD com duas saídas: um alto-falante bluetooth e um servidor web para servir fluxos de áudio para qualquer reprodutor de mídia de streaming.

O Bluetooth costuma ser bastante frustrante de configurar. Você terá que parear seu dispositivo Bluetooth e certificar-se de que o dispositivo seja conectado automaticamente assim que for ligado. O serviço do sistema Bluetooth retornado pelo procedimento `bluetooth-service` fornece a infraestrutura necessária para configurar isso.

Reconfigure seu sistema com pelo menos os seguintes serviços e pacotes:

```
(operating-system
  ;; ...
  (packages (cons* bluez bluez-alsa
                   %base-packages))
  (services
   ;; ...
   (dbus-service #:services (list bluez-alsa))
   (bluetooth-service #:auto-enable? #t)))
```

Inicie o serviço `bluetooth` e então use `bluetoothctl` para escanear dispositivos Bluetooth. Tente identificar seu alto-falante Bluetooth e escolher seu ID de dispositivo na lista resultante de dispositivos que é indubitavelmente dominada por uma desconcertante miscelânea de engenhocas de automação residencial de seus vizinhos. Isso só precisa ser feito uma vez:

```
$ bluetoothctl
[NEW] Controller 00:11:22:33:95:7F BlueZ 5.40 [default]

[bluetooth]# power on
[bluetooth]# Changing power on succeeded

[bluetooth]# agent on
[bluetooth]# Agent registered

[bluetooth]# default-agent
[bluetooth]# Default agent request successful

[bluetooth]# scan on
[bluetooth]# Discovery started
[CHG] Controller 00:11:22:33:95:7F Discovering: yes
[NEW] Device AA:BB:CC:A4:AA:CD My Bluetooth Speaker
[NEW] Device 44:44:FF:2A:20:DC My Neighbor's TV
...
```

```
[bluetooth]# pair AA:BB:CC:A4:AA:CD
Attempting to pair with AA:BB:CC:A4:AA:CD
[CHG] Device AA:BB:CC:A4:AA:CD Connected: yes

[Meu alto-falante Bluetooth]# [CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110b-0000-1000
[CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110c-0000-1000-8000-00xxxxxxxxxx
[CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110e-0000-1000-8000-00xxxxxxxxxx
[CHG] Device AA:BB:CC:A4:AA:CD Paired: yes
Pairing successful

[CHG] Device AA:BB:CC:A4:AA:CD Connected: no

[bluetooth]#
[bluetooth]# trust AA:BB:CC:A4:AA:CD
[bluetooth]# [CHG] Device AA:BB:CC:A4:AA:CD Trusted: yes
Changing AA:BB:CC:A4:AA:CD trust succeeded

[bluetooth]#
[bluetooth]# connect AA:BB:CC:A4:AA:CD
Attempting to connect to AA:BB:CC:A4:AA:CD
[bluetooth]# [CHG] Device AA:BB:CC:A4:AA:CD RSSI: -63
[CHG] Device AA:BB:CC:A4:AA:CD Connected: yes
Connection successful

[My Bluetooth Speaker]# scan off
[CHG] Device AA:BB:CC:A4:AA:CD RSSI is nil
Discovery stopped
[CHG] Controller 00:11:22:33:95:7F Discovering: no
```

Parabéns, agora você pode se conectar automaticamente ao seu alto-falante Bluetooth!

Agora é hora de configurar o ALSA para usar o módulo Bluetooth *bluealsa*, para que você possa definir um dispositivo ALSA pcm correspondente ao seu alto-falante Bluetooth. Para um servidor headless, usar *bluealsa* com um dispositivo Bluetooth fixo é provavelmente mais simples do que configurar o Pulseaudio e seu comportamento de troca de fluxo. Configuramos o ALSA criando um `alsa-configuration` personalizado para o `alsa-service-type`. A configuração declarará um tipo `pcm bluealsa` do módulo `bluealsa` fornecido pelo pacote `bluez-alsa` e, em seguida, definirá um dispositivo `pcm` desse tipo para seu alto-falante Bluetooth.

Tudo o que resta então é fazer o MPD enviar dados de áudio para este dispositivo ALSA. Também adicionamos uma saída MPD secundária que torna os arquivos de áudio atualmente reproduzidos disponíveis como um fluxo por meio de um servidor web na porta 8080. Quando habilitado, um dispositivo na rede pode ouvir o fluxo de áudio conectando qualquer reprodutor de mídia capaz ao servidor HTTP na porta 8080, independentemente do status do alto-falante Bluetooth.

O que se segue é o esboço de uma declaração `operating-system` que deve realizar as tarefas mencionadas acima:

```
(use-modules (gnu))
```

```
(use-service-modules audio dbus sound #;... etc)
(use-package-modules audio linux #;... etc)
(operating-system
  ;; ...
  (packages (cons* bluez bluez-alsa
                   %base-packages))
  (services
   ;; ...
   (service mpd-service-type
            (mpd-configuration
             (user "your-username")
             (music-dir "/path/to/your/music")
             (address "192.168.178.20")
             (outputs (list (mpd-output
                             (type "alsa")
                             (name "MPD")
                             (extra-options
                              ;; Use o mesmo nome que no ALSA
                              ;; configuração abaixo.
                              '((device . "pcm.btspeaker"))))
                            (mpd-output
                             (type "httpd")
                             (name "streaming")
                             (enabled? #false)
                             (always-on? #true)
                             (tags? #true)
                             (mixer-type 'null)
                             (extra-options
                              '((encoder . "vorbis")
                                (port    . "8080")
                                (bind-to-address . "192.168.178.20")
                                (max-clients . "0") ;no limit
                                (quality . "5.0")
                                (format  . "44100:16:1")))))))))
   (dbus-service #:services (list bluez-alsa))
   (bluetooth-service #:auto-enable? #t)
   (service alsa-service-type
            (alsa-configuration
             (pulseaudio? #false) ;nós não precisamos disso
             (extra-options
              #~(string-append "\
# Declarar o tipo de dispositivo de áudio Bluetooth \"bluealsa\" do módulo bluealsa
pcm_type.bluealsa {
    lib \""
#$(file-append bluez-alsa "/lib/alsa-lib/libasound_module_pcm_bluealsa.so") "\"
}
```

```
# Declarar tipo de dispositivo de controle \"bluealsa\" do mesmo módulo
ctl_type.bluealsa {
    lib \""
#$(file-append bluez-alsa "/lib/alsa-lib/libasound_module_ctl_bluealsa.so") "\"
}

# Defina o dispositivo de áudio Bluetooth real.
pcm.btspeaker {
    type bluealsa
    device \"AA:BB:CC:A4:AA:CD\" # identificador de dispositivo exclusivo
    profile \"a2dp\"
}

# Defina um controlador associado.
ctl.btspeaker {
    type bluealsa
}
"))))))
```

Aproveite a música com o cliente MPD de sua escolha ou um media player capaz de transmitir via HTTP!

4 Contêineres

O kernel Linux fornece uma série de facilidades compartilhadas que estão disponíveis para processos no sistema. Essas facilidades incluem uma visão compartilhada no sistema de arquivos, outros processos, dispositivos de rede, identidades de usuários e grupos e alguns outros. Desde o Linux 3.19, um usuário pode escolher *unshare* algumas dessas facilidades compartilhadas para processos selecionados, fornecendo a eles (e seus processos filhos) uma visão diferente do sistema.

Um processo com um namespace `mount` não compartilhado, por exemplo, tem sua própria visão no sistema de arquivos — ele só poderá ver diretórios que foram explicitamente vinculados em seu namespace mount. Um processo com seu próprio namespace `proc` se considerará o único processo em execução no sistema, executando como PID 1.

O Guix usa esses recursos do kernel para fornecer ambientes totalmente isolados e até mesmo contêineres Guix System completos, máquinas virtuais leves que compartilham o kernel do sistema host. Esse recurso é especialmente útil ao usar o Guix em uma distribuição estrangeira para evitar interferência de bibliotecas estrangeiras ou arquivos de configuração que estão disponíveis em todo o sistema.

4.1 Contêineres Guix

A maneira mais fácil de começar é usar `guix shell` com a opção `--container`. Veja Seção "Invocando guix shell" em *Manual de Referência GNU Guix* para uma referência de opções válidas.

O snippet a seguir gera um processo de shell mínimo com a maioria dos namespaces não compartilhados do sistema. O diretório de trabalho atual é visível para o processo, mas qualquer outra coisa no sistema de arquivos não está disponível. Esse isolamento extremo pode ser muito útil quando você deseja descartar qualquer tipo de interferência de variáveis de ambiente, bibliotecas instaladas globalmente ou arquivos de configuração.

```
guix shell --container
```

O trecho a seguir gera um processo de shell mínimo como a maioria dos namespaces não compartilhados do sistema. O diretório de trabalho atual é visível para o processo, mas qualquer outra coisa no sistema de arquivos não está disponível. Esse extremo de isolamento pode ser muito útil quando você deseja descartar qualquer tipo de interferência de variáveis de ambiente, bibliotecas instaladas globalmente ou arquivos de configuração.

```
$ echo /gnu/store/*
/gnu/store/...-gcc-10.3.0-lib
/gnu/store/...-glibc-2.33
/gnu/store/...-bash-static-5.1.8
/gnu/store/...-ncurses-6.2.20210619
/gnu/store/...-bash-5.1.8
/gnu/store/...-profile
/gnu/store/...-readline-8.1.1
```

Não há muito que você possa fazer em um ambiente como esse além de sair dele. Você pode usar `^D` ou `exit` para encerrar esse ambiente de shell limitado.

Você pode tornar outros diretórios disponíveis dentro do ambiente do contêiner; use `--expose=DIRECTORY` para montar o diretório fornecido como um local somente leitura

dentro do contêiner, ou use `--share=DIRECTORY` para tornar o local gravável. Com um argumento de mapeamento adicional após o nome do diretório, você pode controlar o nome do diretório dentro do contêiner. No exemplo a seguir, mapeamos `/etc` no sistema host para `/the/host/etc` dentro de um contêiner no qual os GNU coreutils estão instalados.

```
$ guix shell --container --share=/etc=/the/host/etc coreutils
$ ls /the/host/etc
```

Da mesma forma, você pode evitar que o diretório de trabalho atual seja mapeado para o contêiner com a opção `--no-cwd`. Outra boa ideia é criar um diretório dedicado que servirá como o diretório home do contêiner e gerar o shell do contêiner a partir desse diretório.

Em um sistema estrangeiro, um ambiente de contêiner pode ser usado para compilar software que não pode ser vinculado a bibliotecas do sistema ou à cadeia de ferramentas do compilador do sistema. Um caso de uso comum em um contexto de pesquisa é instalar pacotes de dentro de uma sessão R. Fora de um ambiente de contêiner, há uma boa chance de que a cadeia de ferramentas do compilador estrangeiro e bibliotecas de sistema incompatíveis sejam encontradas primeiro, resultando em binários incompatíveis que não podem ser usados pelo R. Em um shell de contêiner, esse problema desaparece, pois as bibliotecas e executáveis do sistema simplesmente não estão disponíveis devido ao namespace `mount` não compartilhado.

Vamos pegar um manifesto abrangente que fornece um ambiente de desenvolvimento confortável para uso com R:

```
(specifications->manifest
  (list "r-minimal"

        ;; base packages
        "bash-minimal"
        "glibc-locales"
        "nss-certs"

        ;; Ferramentas comuns de linha de comando para o caso de o contêiner ficar mui
        "coreutils"
        "grep"
        "which"
        "wget"
        "sed"

        ;; Ferramenta de marcação R
        "pandoc"

        ;; Cadeia de ferramentas e bibliotecas comuns para "install.packages"
        "gcc-toolchain@10"
        "gfortran-toolchain"
        "gawk"
        "tar"
        "gzip"
        "unzip"
        "make"
```

```
        "cmake"
        "pkg-config"
        "cairo"
        "libxt"
        "openssl"
        "curl"
        "zlib"))
```

Vamos usar isso para executar o R dentro de um ambiente de contêiner. Por conveniência, compartilhamos o namespace `net` para usar as interfaces de rede do sistema host. Agora podemos construir pacotes R a partir da fonte da maneira tradicional, sem ter que nos preocupar com incompatibilidades ou incompatibilidades de ABI.

```
$ guix shell --container --network --manifest=manifest.scm -- R

R version 4.2.1 (2022-06-23) -- "Funny-Looking Kid"
Copyright (C) 2022 The R Foundation for Statistical Computing
...
> e <- Sys.getenv("GUIX_ENVIRONMENT")
> Sys.setenv(GIT_SSL_CAINFO=paste0(e, "/etc/ssl/certs/ca-certificates.crt"))
> Sys.setenv(SSL_CERT_FILE=paste0(e, "/etc/ssl/certs/ca-certificates.crt"))
> Sys.setenv(SSL_CERT_DIR=paste0(e, "/etc/ssl/certs"))
> install.packages("Cairo", lib=paste0(getwd()))
...
* installing *source* package 'Cairo' ...
...
* DONE (Cairo)

The downloaded source packages are in
'/tmp/RtmpCuwdwM/downloaded_packages'
> library("Cairo", lib=getwd())
> # sucesso!
```

Usar shells de contêiner é divertido, mas eles podem se tornar um pouco trabalhosos quando você quer ir além de apenas um único processo interativo. Algumas tarefas se tornam muito mais fáceis quando se assentam na fundação sólida de um Sistema Guix adequado e seu rico conjunto de serviços de sistema. A próxima seção mostra como iniciar um Sistema Guix completo dentro de um contêiner.

4.2 Contêineres do Sistema Guix

O Guix System fornece uma ampla gama de serviços de sistema interconectados que são configurados declarativamente para formar uma fundação GNU System sem estado confiável para quaisquer tarefas que você jogue nele. Mesmo ao usar o Guix em uma distribuição estrangeira, você pode se beneficiar do design do Guix System executando uma instância do sistema como um contêiner. Usando os mesmos recursos do kernel de namespaces não compartilhados mencionados na seção anterior, a instância do Guix System resultante é isolada do sistema host e compartilha apenas os locais do sistema de arquivos que você declara explicitamente.

Um contêiner do Sistema Guix difere do processo shell criado por `guix shell --container` em várias maneiras importantes. Enquanto em um shell de contêiner o processo containerizado é um processo de shell Bash, um contêiner do Sistema Guix executa o Shepherd como PID 1. Em um contêiner do sistema, todos os serviços do sistema (veja Seção "Serviços" em *Manual de Referência do GNU Guix*) são configurados exatamente como seriam em um Sistema Guix em uma máquina virtual ou em bare metal — isso inclui daemons gerenciados pelo GNU Shepherd (veja Seção "Serviços do Shepherd" em *Manual de Referência do GNU Guix*), bem como outros tipos de extensões para o sistema operacional (veja Seção "Composição de serviço" em *Manual de Referência do GNU Guix*).

O aumento percebido na complexidade da execução de um contêiner do Guix System é facilmente justificado ao lidar com aplicativos mais complexos que têm requisitos mais altos ou mais rígidos em seus contextos de execução: arquivos de configuração, contas de usuários dedicadas, diretórios para caches ou arquivos de log, etc. No Guix System, as demandas desse tipo de software são satisfeitas por meio da implantação de serviços do sistema.

4.2.1 Um banco de dados de contêineres

Um bom exemplo pode ser um servidor de banco de dados PostgreSQL. Grande parte da complexidade de configurar tal servidor de banco de dados está encapsulada nesta declaração de serviço enganosamente curta:

```
(service postgresql-service-type
         (postgresql-configuration
          (postgresql postgresql-14)))
```

Uma declaração completa do sistema operacional para uso com um contêiner do sistema Guix seria algo como isto:

```
(use-modules (gnu))
(use-package-modules databases)
(use-service-modules databases)

(operating-system
  (host-name "container")
  (timezone "Europe/Berlin")
  (file-systems (cons (file-system
                        (device (file-system-label "does-not-matter"))
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))
  (bootloader (bootloader-configuration
               (bootloader grub-bootloader)
               (targets '("/dev/sdX"))))
  (services
   (cons* (service postgresql-service-type
                   (postgresql-configuration
                    (postgresql postgresql-14)
                    (config-file
                     (postgresql-config-file
```

```
                      (log-destination "stderr")
                      (hba-file
                       (plain-file "pg_hba.conf"
                                   "\
local all all trust
host all all 10.0.0.1/32  trust"))
                      (extra-config
                       '(("listen_addresses" "*")
                         ("log_directory"    "/var/log/postgresql")))))))
          (service postgresql-role-service-type
                   (postgresql-role-configuration
                    (roles
                     (list (postgresql-role
                            (name "test")
                            (create-database? #t))))))
          %base-services)))
```

Com `postgresql-role-service-type` definimos uma função "test" e criamos um banco de dados correspondente, para que possamos testar imediatamente sem nenhuma configuração manual adicional. As configurações do `postgresql-config-file` permitem que um cliente do endereço IP 10.0.0.1 se conecte sem exigir autenticação — uma má ideia em sistemas de produção, mas conveniente para este exemplo.

Vamos construir um script que irá iniciar uma instância deste Guix System como um contêiner. Escreva a declaração `operating-system` acima em um arquivo `os.scm` e então use `guix system container` para construir o inicializador. (veja Seção "Invocando guix system" em *Manual de Referência do GNU Guix*).

```
$ guix system container os.scm
As seguintes derivações serão compiladas::
/gnu/store/...-run-container.drv
...
compilando /gnu/store/...-run-container.drv...
/gnu/store/...-run-container
```

Agora que temos um script de launcher, podemos executá-lo para gerar o novo sistema com um serviço PostgreSQL em execução. Observe que, devido a algumas limitações ainda não resolvidas, precisamos executar o launcher como usuário root, por exemplo, com `sudo`.

```
$ sudo /gnu/store/...-run-container
O contêiner de sistema está rodando como PID 5983
...
```

Coloque o processo em segundo plano com `Ctrl-z` seguido por `bg`. Observe o ID do processo na saída; precisaremos dele para conectar ao contêiner mais tarde. Quer saber? Vamos tentar anexar ao contêiner agora mesmo. Usaremos `nsenter`, uma ferramenta fornecida pelo pacote `util-linux`:

```
$ guix shell util-linux
$ sudo nsenter -a -t 5983
root@container /# pgrep -a postgres
49 /gnu/store/...-postgresql-14.4/bin/postgres -D /var/lib/postgresql/data --config-fi
```

```
51 postgres: checkpointer
52 postgres: background writer
53 postgres: walwriter
54 postgres: autovacuum launcher
55 postgres: stats collector
56 postgres: logical replication launcher
root@container /# exit
```

O serviço PostgreSQL está sendo executado no contêiner!

4.2.2 Rede em contêineres

De que adianta um Sistema Guix rodando um serviço de banco de dados PostgreSQL como um contêiner quando só podemos falar com ele com processos originados no contêiner? Seria muito melhor se pudéssemos falar com o banco de dados pela rede.

A maneira mais fácil de fazer isso é criar um par de dispositivos Ethernet virtuais conectados (conhecidos como `veth`). Movemos um dos dispositivos (`ceth-test`) para o namespace `net` do contêiner e deixamos a outra extremidade (`veth-test`) da conexão no sistema host.

```
pid=5983
ns="guix-test"
host="veth-test"
client="ceth-test"

# Anexe o novo namespace de rede "guix-test" ao PID do contêiner.
sudo ip netns attach $ns $pid

# Crie o par de dispositivos
sudo ip link add $host type veth peer name $client

# Mova o dispositivo cliente para o namespace de rede do contêiner
sudo ip link set $client netns $ns
```

Em seguida, configuramos o lado do host:

```
sudo ip link set $host up
sudo ip addr add 10.0.0.1/24 dev $host
```

. . . e então configuramos o lado do cliente:

```
sudo ip netns exec $ns  ip link set lo up
sudo ip netns exec $ns  ip link set $client up
sudo ip netns exec $ns  ip addr add 10.0.0.2/24 dev $client
```

Neste ponto, o host pode alcançar o contêiner no endereço IP 10.0.0.2, e o contêiner pode alcançar o host no IP 10.0.0.1. Isso é tudo o que precisamos para falar com o servidor de banco de dados dentro do contêiner do sistema host do lado de fora.

```
$ psql -h 10.0.0.2 -U test
psql (14.4)
Type "help" for help.

test=> CREATE TABLE hello (who TEXT NOT NULL);
CREATE TABLE
```

```
test=> INSERT INTO hello (who) VALUES ('world');
INSERT 0 1
test=> SELECT * FROM hello;
  who
-------
 world
(1 row)
```

Agora que terminamos esta pequena demonstração, vamos limpar:

```
sudo kill $pid
sudo ip netns del $ns
sudo ip link del $host
```

5 Máquinas Virtuais

O Guix pode produzir imagens de disco (veja Seção "Invocando guix system" em *Manual de Referência do GNU Guix*) que podem ser usadas com soluções de máquinas virtuais como virt-manager, GNOME Boxes ou o mais simples QEMU, entre outros.

Este capítulo tem como objetivo fornecer exemplos práticos e práticos relacionados ao uso e à configuração de máquinas virtuais em um sistema Guix.

5.1 Ponte de rede para QEMU

Por padrão, o QEMU usa um back-end de rede host chamado "modo usuário", o que é conveniente, pois não requer nenhuma configuração. Infelizmente, também é bastante limitado. Neste modo, o convidado VM (máquina virtual) pode acessar a rede da mesma forma que o host, mas não pode ser alcançado a partir do host. Além disso, como o modo de rede do usuário do QEMU depende do ICMP, ferramentas de rede baseadas em ICMP, como `ping`, *não* funcionam neste modo. Portanto, geralmente é desejável configurar uma ponte de rede, que permite que o convidado participe totalmente da rede. Isso é necessário, por exemplo, quando o convidado deve ser usado como um servidor.

5.1.1 Criando uma interface de ponte de rede

Há muitas maneiras de criar uma ponte de rede. O comando a seguir mostra como usar o NetworkManager e sua ferramenta de interface de linha de comando (CLI) `nmcli`, que já deve estar disponível se a declaração do seu sistema operacional for baseada em um dos modelos de desktop:

```
# nmcli con add type bridge con-name br0 ifname br0
```

Para que essa ponte faça parte da sua rede, você deve associar sua ponte de rede à interface Ethernet usada para conectar-se à rede. Supondo que sua interface seja chamada '`enp2s0`', o comando a seguir pode ser usado para fazer isso:

```
# nmcli con add type bridge-slave ifname enp2s0 master br0
```

Importante: Somente interfaces Ethernet podem ser adicionadas a uma ponte. Para interfaces sem fio, considere a abordagem de rede roteada detalhada em Veja Seção 5.2 [Roteamento de rede para libvirt], Página 58.

Por padrão, a ponte de rede permitirá que seus convidados obtenham seus endereços IP via DHCP, se disponível em sua rede local. Para simplificar, é isso que usaremos aqui. Para encontrar facilmente os convidados, eles podem ser configurados para anunciar seus nomes de host via mDNS.

5.1.2 Configurando o script auxiliar da ponte QEMU

O QEMU vem com um programa auxiliar para usar convenientemente uma interface de ponte de rede como um usuário sem privilégios veja Seção "Network options" em *Documentação do QEMU*. O binário deve ser definido como setuid root para operação adequada; isso pode ser obtido adicionando-o ao campo `privileged-programs` da sua definição `operating-system` (host), conforme mostrado abaixo:

```
(privileged-programs
 (cons (privileged-program
```

```
          (program (file-append qemu "/libexec/qemu-bridge-helper"))
          (setuid? #t))
        %default-privileged-programs))
```

O arquivo `/etc/qemu/bridge.conf` também deve ser feito para permitir a interface bridge, já que o padrão é negar tudo. Adicione o seguinte à sua lista de serviços para fazer isso:

```
(extra-special-file "/etc/qemu/host.conf" "allow br0\n")
```

5.1.3 Invocando QEMU com as opções de linha de comando corretas

Ao invocar o QEMU, as seguintes opções devem ser fornecidas para que a ponte de rede seja usada, após selecionar um endereço MAC exclusivo para o convidado.

> **Importante:** Por padrão, um único endereço MAC é usado para todos os convidados, a menos que seja fornecido. Deixar de fornecer endereços MAC diferentes para cada máquina virtual que faz uso da ponte causaria problemas de rede.

```
$ qemu-system-x86_64 [...] \
    -device virtio-net-pci,netdev=user0,mac=XX:XX:XX:XX:XX:XX \
    -netdev bridge,id=user0,br=br0 \
    [...]
```

Para gerar endereços MAC que tenham o prefixo registrado QEMU, o seguinte snippet pode ser empregado:

```
mac_address="52:54:00:$(dd if=/dev/urandom bs=512 count=1 2>/dev/null \
                          | md5sum \
                          | sed -E 's/^(..)(..)(..).*$/\1:\2:\3/')"
echo $mac_address
```

5.1.4 Problemas de rede causados pelo Docker

Se você usa o Docker na sua máquina, você pode ter problemas de conectividade ao tentar usar uma ponte de rede, que são causados pelo Docker também depender de pontes de rede e configurar suas próprias regras de roteamento. A solução é adicionar o seguinte snippet `iptables` à sua declaração `operating-system`:

```
(service iptables-service-type
             (iptables-configuration
              (ipv4-rules (plain-file "iptables.rules" "\
*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A FORWARD -i br0 -o br0 -j ACCEPT
COMMIT
"))
```

5.2 Roteamento de rede para libvirt

Se a máquina que hospeda suas máquinas virtuais estiver conectada sem fio à rede, você não poderá usar uma ponte de rede verdadeira, conforme explicado na seção anterior (veja

Seção 5.1 [Ponte de rede para QEMU], Página 57). Nesse caso, a próxima melhor opção é usar uma ponte *virtual* com roteamento estático e configurar uma máquina virtual com libvirt para usá-la (por meio da GUI `virt-manager`, por exemplo). Isso é semelhante ao modo de operação padrão do QEMU/libvirt, exceto que, em vez de usar NAT (Network Address Translation), ele depende de rotas estáticas para unir o endereço IP da VM (máquina virtual) à LAN (rede local). Isso fornece conectividade bidirecional de e para a máquina virtual, necessária para expor serviços hospedados na máquina virtual.

5.2.1 Criando uma ponte de rede virtual

Uma ponte de rede virtual consiste em alguns componentes/configurações, como uma interface TUN (túnel de rede), servidor DHCP (dnsmasq) e regras de firewall (iptables). O comando `virsh`, fornecido pelo pacote `libvirt`, torna muito fácil criar uma ponte virtual. Primeiro, você precisa escolher uma sub-rede de rede para sua ponte virtual; se sua LAN doméstica estiver na rede '`192.168.1.0/24`', você pode optar por usar, por exemplo, '`192.168.2.0/24`'. Defina um arquivo XML, por exemplo, `/tmp/virbr0.xml`, contendo o seguinte:

```
<network>
  <name>virbr0</name>
  <bridge name="virbr0" />
  <forward mode="route"/>
  <ip address="192.168.2.0" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.2.1" end="192.168.2.254"/>
    </dhcp>
  </ip>
</network>
```

Em seguida, crie e configure a interface usando o comando `virsh`, como root:

```
virsh net-define /tmp/virbr0.xml
virsh net-autostart virbr0
virsh net-start virbr0
```

A interface '`virbr0`' agora deve estar visível, por exemplo, via o comando '`ip address`'. Ela será iniciada automaticamente toda vez que sua máquina virtual libvirt for iniciada.

5.2.2 Configurando as rotas estáticas para sua ponte virtual

Se você configurou sua máquina virtual para usar sua interface de ponte virtual '`virbr0`' recém-criada, ela já deve receber um IP via DHCP, como '`192.168.2.15`' e ser acessível a partir do servidor que a hospeda, por exemplo, via '`ping 192.168.2.15`'. Há uma última configuração necessária para que a VM possa alcançar a rede externa: adicionar rotas estáticas ao roteador da rede.

Neste exemplo, a rede LAN é '`192.168.1.0/24`' e a página da web de configuração do roteador pode ser acessada por meio de, por exemplo, a página `http://192.168.1.1`. Em um roteador executando o firmware libreCMC (`https://librecmc.org/`), você navegaria até a página Network → Static Routes (`https://192.168.1.1/cgi-bin/luci/admin/network/routes`) e adicionaria uma nova entrada em '`Static IPv4 Routes`' com as seguintes informações:

‘`Interface`’
: rede local

‘`Alvo`’ 192.168.2.0

‘`Rede-IPV4`’
: 255.255.255.0

‘`IPv4-Ponto de entrada`’
: *servidor-ip*

‘`Tipo de rota`’
: unicast

onde *server-ip* é o endereço IP da máquina que hospeda as VMs, que deve ser estático.

Depois de salvar/aplicar essa nova rota estática, a conectividade externa deve funcionar de dentro da sua VM; você pode, por exemplo, executar ‘`ping gnu.org`’ para verificar se ela funciona corretamente.

6 Gerenciamento avançado de pacotes

Guix é um gerenciador de pacotes funcional que oferece muitos recursos além do que os gerenciadores de pacotes mais tradicionais podem fazer. Para os não iniciados, esses recursos podem não ter casos de uso óbvios a princípio. O propósito deste capítulo é demonstrar alguns conceitos avançados de gerenciamento de pacotes.

veja Seção "Gerenciamento de pacotes" em *Manual de Referência GNU Guix* para uma referência completa.

6.1 Perfis Guix na Prática

Guix fornece um recurso muito útil que pode ser bem estranho para novatos: *profiles*. Eles são uma maneira de agrupar instalações de pacotes e todos os usuários no mesmo sistema são livres para usar quantos perfis quiserem.

Seja você um desenvolvedor ou não, você pode descobrir que múltiplos perfis trazem grande poder e flexibilidade. Embora eles mudem um pouco o paradigma em comparação aos *gerenciadores de pacotes tradicionais*, eles são muito convenientes de usar depois que você entende como configurá-los.

> **Nota:** Esta seção é um guia opinativo sobre o uso de múltiplos perfis. Ele é anterior ao `guix shell` e seu cache de perfil rápido (veja Seção "Invocando guix shell" em *Manual de Referência do GNU Guix*).
>
> Em muitos casos, você pode descobrir que usar `guix shell` para configurar o ambiente que você precisa, quando você precisa, dá menos trabalho do que manter um perfil dedicado. Sua escolha!

Se você estiver familiarizado com o '`virtualenv`' do Python, você pode pensar em um perfil como um tipo de '`virtualenv`' universal que pode conter qualquer tipo de software, não apenas software Python. Além disso, os perfis são autossuficientes: eles capturam todas as dependências de tempo de execução, o que garante que todos os programas dentro de um perfil sempre funcionarão em qualquer ponto do tempo.

Perfis múltiplos têm muitos benefícios:

- Separação semântica limpa dos vários pacotes que um usuário precisa para diferentes contextos.
- Vários perfis podem ser disponibilizados no ambiente no login ou em um shell dedicado.
- Os perfis podem ser carregados sob demanda. Por exemplo, o usuário pode usar vários shells, cada um deles executando perfis diferentes.
- Isolamento: programas de um perfil não usarão programas do outro, e o usuário pode até instalar versões diferentes dos mesmos programas nos dois perfis sem conflito.
- Deduplicação: Perfis compartilham dependências que são exatamente as mesmas. Isso torna o armazenamento de múltiplos perfis eficiente.
- Reproduzível: quando usado com manifestos declarativos, um perfil pode ser totalmente especificado pelo commit do Guix que estava ativo quando foi configurado. Isso significa que o mesmo perfil exato pode ser configurado em qualquer lugar e a qualquer hora (`https://guix.gnu.org/blog/2018/multi-dimensional-transactions-and-rollbacks-oh-my/`), com apenas as

informações de commit. Veja a seção sobre Seção 6.1.5 [Perfis reproduzíveis], Página 65.

- Atualizações e manutenção mais fáceis: vários perfis facilitam a manutenção de listas de pacotes em mãos e tornam as atualizações completamente sem atrito.

Concretamente, seguem alguns perfis típicos:

- As dependências de um projeto no qual você está funcionando.
- Suas bibliotecas de linguagens de programação favoritas.
- Programas específicos para laptop (como '`powertop`') que você não precisa em um desktop.
- TEXlive (este pode ser muito útil quando você precisa instalar apenas um pacote para este documento que acabou de receber por e-mail).
- Jogos.

Vamos mergulhar na configuração!

6.1.1 Configuração básica com manifestos

Um perfil Guix pode ser configurado *via* um *manifest*. Um manifesto é um trecho de código Scheme que especifica o conjunto de pacotes que você quer ter em seu perfil; parece com isso:

```
(specifications->manifest
 '("pacote-1"
   ;; Versão 1.3 do pacote-2.
   "pacote-2@1.3"
   ;; A saída "lib" do pacote-3.
   "pacote-3:lib"
   ; ...
   "pacote-N"))
```

Veja Seção "Escrevendo manifestos" em *Manual de referência do GNU Guix*, para mais informações sobre a sintaxe.

Podemos criar uma especificação de manifesto por perfil e instalá-los desta maneira:

```
GUIX_EXTRA_PROFILES=$HOME/.guix-extra-profiles
mkdir -p "$GUIX_EXTRA_PROFILES"/meu-projeto # se ainda não existir
guix package --manifest=/caminho/para/guix-meu-projeto-manifest.scm \
 --profile="$GUIX_EXTRA_PROFILES"/meu-projeto/meu-projeto
```

Aqui definimos uma variável arbitrária '`GUIX_EXTRA_PROFILES`' para apontar para o diretório onde armazenaremos nossos perfis no restante deste artigo.

Colocar todos os seus perfis em um único diretório, com cada perfil recebendo seu próprio subdiretório, é um pouco mais limpo. Dessa forma, cada subdiretório conterá todos os links simbólicos para precisamente um perfil. Além disso, "fazer loop sobre perfis" se torna óbvio em qualquer linguagem de programação (por exemplo, um script de shell) simplesmente fazendo loop sobre os subdiretórios de '`$GUIX_EXTRA_PROFILES`'.

Observe que também é possível fazer um loop na saída de

```
guix package --list-profiles
```

embora você provavelmente tenha que filtrar `~/.config/guix/current`.

Para habilitar todos os perfis no login, adicione isto ao seu `~/.bash_profile` (ou similar):

```
for i in $GUIX_EXTRA_PROFILES/*; do
  profile=$i/$(basename "$i")
  if [ -f "$profile"/etc/profile ]; then
    GUIX_PROFILE="$profile"
    . "$GUIX_PROFILE"/etc/profile
  fi
  unset profile
done
```

Nota para usuários do sistema Guix: o acima reflete como seu perfil padrão `~/.guix-profile` é ativado a partir de `/etc/profile`, sendo este último carregado por `~/.bashrc` por padrão.

Obviamente, você pode escolher habilitar apenas um subconjunto deles:

```
for i in "$GUIX_EXTRA_PROFILES"/meu-projeto-1 "$GUIX_EXTRA_PROFILES"/meu-projeto-2; do
  profile=$i/$(basename "$i")
  if [ -f "$profile"/etc/profile ]; then
    GUIX_PROFILE="$profile"
    . "$GUIX_PROFILE"/etc/profile
  fi
  unset profile
done
```

Quando um perfil está desativado, é fácil habilitá-lo para um shell individual sem "poluir" o restante da sessão do usuário:

```
GUIX_PROFILE="path/to/my-project" ; . "$GUIX_PROFILE"/etc/profile
```

A chave para habilitar um perfil é *source* seu arquivo '`etc/profile`'. Este arquivo contém código shell que exporta as variáveis de ambiente corretas necessárias para ativar o software contido no perfil. Ele é construído automaticamente pelo Guix e deve ser sourced. Ele contém as mesmas variáveis que você obteria se executasse:

```
guix package --search-paths=prefix --profile=$my_profile"
```

Once again, see Seção "Invoking guix package" em *GNU Guix Reference Manual* for the command line options.

Para atualizar um perfil, basta instalar o manifesto novamente:

```
guix package -m /caminho/para/guix-meu-projeto-manifest.scm \
 -p "$GUIX_EXTRA_PROFILES"/meu-projeto/meu-projeto
```

Para atualizar todos os perfis, é fácil fazer um loop sobre eles. Por exemplo, supondo que suas especificações de manifesto estejam armazenadas em `~/.guix-manifests/guix-$profile-manifest.scm`, com '`$profile`' sendo o nome do perfil (por exemplo, "project1"), você pode fazer o seguinte no Bourne shell:

```
for profile in "$GUIX_EXTRA_PROFILES"/*; do
  guix package --profile="$profile" \
   --manifest="$HOME/.guix-manifests/guix-$profile-manifest.scm"
done
```

Cada perfil tem suas próprias gerações:

```
guix package -p "$GUIX_EXTRA_PROFILES"/meu-projeto/meu-projeto --list-generations
```

Você pode reverter para qualquer geração de um determinado perfil:

```
guix package -p "$GUIX_EXTRA_PROFILES"/meu-projeto/meu-projeto --switch-generations=17
```

Por fim, se você quiser alternar para um perfil sem herdar do ambiente atual, poderá ativá-lo a partir de um shell vazio:

```
env -i $(which bash) --login --noprofile --norc
. meu-projeto/etc/profile
```

6.1.2 Pacotes necessários

Ativar um perfil basicamente se resume a exportar um monte de variáveis ambientais. Essa é a função do '`etc/profile`' dentro do perfil.

Nota: Somente as variáveis de ambiente dos pacotes que os consomem serão definidas.

Por exemplo, '`MANPATH`' não será definido se não houver um aplicativo de consumidor para páginas de manual dentro do perfil. Então, se você precisar acessar páginas de manual de forma transparente depois que o perfil for carregado, você tem duas opções:

- Exporte a variável manualmente, por exemplo

  ```
  export MANPATH=/caminho/para/perfil${MANPATH:+:}$MANPATH
  ```

- Ou inclua '`man-db`' no manifesto do perfil.

O mesmo vale para '`INFOPATH`' (você pode instalar '`info-reader`'), '`PKG_CONFIG_PATH`' (instale '`pkg-config`'), etc.

6.1.3 Perfil padrão

E quanto ao perfil padrão que o Guix mantém em `~/.guix-profile`?

Você pode atribuir a ele o papel que quiser. Normalmente, você instalaria o manifesto dos pacotes que deseja usar o tempo todo.

Alternativamente, você pode mantê-lo "sem manifesto" para pacotes descartáveis que você usaria apenas por alguns dias. Dessa forma, é conveniente executar

```
guix install package-foo
guix upgrade package-bar
```

sem precisar especificar o caminho para um perfil.

6.1.4 Os benefícios dos manifestos

Manifestos permitem que você *declare* o conjunto de pacotes que você gostaria de ter em um perfil (veja Seção "Escrevendo manifestos" em *Manual de Referência do GNU Guix*). Eles são uma maneira conveniente de manter suas listas de pacotes por perto e, digamos, sincronizá-las em várias máquinas usando um sistema de controle de versão.

Uma reclamação comum sobre manifestos é que eles podem ser lentos para instalar quando contêm um grande número de pacotes. Isso é especialmente trabalhoso quando você só quer obter uma atualização para um pacote dentro de um grande manifesto.

Esse é mais um motivo para usar múltiplos perfis, que são perfeitos para dividir manifestos em múltiplos conjuntos de pacotes semanticamente conectados. Usar múltiplos perfis pequenos fornece mais flexibilidade e usabilidade.

Os manifestos vêm com múltiplos benefícios. Em particular, eles facilitam a manutenção:

- Quando um perfil é configurado a partir de um manifesto, o manifesto em si é autossuficiente para manter uma "listagem de pacotes" por perto e reinstalar o perfil mais tarde ou em um sistema diferente. Para perfis ad-hoc, precisaríamos gerar uma especificação de manifesto manualmente e manter as versões de pacote para os pacotes que não usam a versão padrão.
- `guix package --upgrade` sempre tenta atualizar os pacotes que propagaram entradas, mesmo que não haja nada a fazer. Os manifestos Guix removem esse problema.
- Ao atualizar parcialmente um perfil, podem surgir conflitos (devido a dependências divergentes entre os pacotes atualizados e não atualizados) e eles podem ser irritantes para resolver manualmente. Manifestos removem esse problema completamente, já que todos os pacotes são sempre atualizados de uma vez.
- Conforme mencionado acima, manifestos permitem perfis reproduzíveis, enquanto os imperativos `guix install`, `guix upgrade`, etc. não, pois eles produzem perfis diferentes a cada vez, mesmo quando contêm os mesmos pacotes. Veja a discussão relacionada sobre o assunto (`https://issues.guix.gnu.org/issue/33285`).
- As especificações do manifesto podem ser usadas por outros comandos '`guix`'. Por exemplo, você pode executar `guix weather -m manifest.scm` para ver quantos substitutos estão disponíveis, o que pode ajudar você a decidir se quer tentar atualizar hoje ou esperar um pouco. Outro exemplo: você pode executar `guix pack -m manifest.scm` para criar um pacote contendo todos os pacotes no manifesto (e suas referências transitivas).
- Finalmente, manifestos têm uma representação Scheme, o tipo de registro '`<manifest>`'. Eles podem ser manipulados no Scheme e passados para as várias Guix APIs (`https://pt.wikipedia.org/wiki/Interface_de_programa%C3%A7%C3%A3o_de_aplica%C3%A7%C3%B5es`).

É importante entender que, embora manifestos possam ser usados para declarar perfis, eles não são estritamente equivalentes: perfis têm o efeito colateral de "fixar" pacotes no armazém, o que os impede de serem coletados como lixo (veja Seção "Invocando guix gc" em *Manual de Referência do GNU Guix*) e garante que eles ainda estarão disponíveis em qualquer ponto no futuro. O comando `guix shell` também protege perfis usados recentemente da coleta de lixo; perfis que não foram usados por um tempo podem ser coletados como lixo, junto com os pacotes aos quais eles se referem.

Para ter 100% de certeza de que um determinado perfil nunca será coletado, instale o manifesto em um perfil e use `GUIX_PROFILE=/o/perfil; . "$GUIX_PROFILE"/etc/profile` conforme explicado acima: isso garante que nosso ambiente de hacking estará disponível o tempo todo.

Aviso de segurança: Embora manter perfis antigos possa ser conveniente, tenha em mente que pacotes desatualizados podem não ter recebido as correções de segurança mais recentes.

6.1.5 Perfis reproduzíveis

Para reproduzir um perfil bit por bit, precisamos de duas informações:

- um manifesto (veja Seção "Escrevendo manifestos" em *Manual de referência do GNU Guix*);
- uma especificação de canal Guix (veja Seção "Replicando Guix" em *Manual de referência do GNU Guix*).

De fato, manifestos por si só podem não ser suficientes: diferentes versões do Guix (ou diferentes canais) podem produzir saídas diferentes para um determinado manifesto.

Você pode emitir a especificação do canal Guix com '`guix describe --format=channels`' (veja Seção "Invocando guix describe" em *Manual de referência do GNU Guix*). Salve isso em um arquivo, digamos '`channel-specs.scm`'.

Em outro computador, você pode usar o arquivo de especificação de canal e o manifesto para reproduzir exatamente o mesmo perfil:

```
GUIX_EXTRA_PROFILES=$HOME/.guix-extra-profiles
GUIX_EXTRA=$HOME/.guix-extra

mkdir -p "$GUIX_EXTRA"/meu-projeto
guix pull --channels=channel-specs.scm --profile="$GUIX_EXTRA/meu-projeto/guix"

mkdir -p "$GUIX_EXTRA_PROFILES/meu-projeto"
"$GUIX_EXTRA"/meu-projeto/guix/bin/guix package \
 --manifest=/caminho/para/guix-meu-projeto-manifest.scm \
 --profile="$GUIX_EXTRA_PROFILES"/meu-projeto/meu-projeto
```

É seguro excluir o perfil de canal Guix que você acabou de instalar com a especificação do canal, o perfil do projeto não depende dele.

7 Desenvolvimento de software X

Guix é uma ferramenta útil para desenvolvedores; `guix shell`, em particular, fornece um ambiente de desenvolvimento autônomo para seu pacote, não importa em qual(is) idioma(s) ele foi escrito (veja Seção "Invocando guix shell" em *Manual de referência do GNU Guix*). Para se beneficiar dele, você tem que inicialmente escrever uma definição de pacote e tê-la no Guix propriamente dito, ou em um canal, ou diretamente na árvore de código-fonte do seu projeto como um arquivo `guix.scm`. Esta última opção é atraente: tudo o que os desenvolvedores precisam fazer para configurar é clonar o repositório do projeto e executar `guix shell`, sem argumentos.

As necessidades de desenvolvimento vão além dos ambientes de desenvolvimento. Como os desenvolvedores podem realizar a integração contínua de seu código em ambientes de construção Guix? Como eles podem entregar seu código diretamente para usuários aventureiros? Este capítulo descreve um conjunto de arquivos que os desenvolvedores podem adicionar ao seu repositório para configurar ambientes de desenvolvimento baseados em Guix, integração contínua e entrega contínua — tudo de uma vez[1].

7.1 Começando

Como fazemos para "Guixificar" um repositório? O primeiro passo, como vimos, será adicionar um `guix.scm` na raiz do repositório em questão. Usaremos Guile (`https://www.gnu.org/software/guile`) como exemplo neste capítulo: ele é escrito em Scheme (principalmente) e C, e tem várias dependências — uma cadeia de ferramentas de compilação C, bibliotecas C, Autoconf e seus amigos, LaTeX, e assim por diante. O `guix.scm` resultante se parece com a definição de pacote usual (veja Seção "Definindo pacotes" em *Manual de Referência GNU Guix*), só que sem o bit `define-public`:

```
;; O arquivo ‘guix.scm’ para Guile, para uso pelo ‘guix shell’.

(use-modules (guix)
             (guix build-system gnu)
             ((guix licenses) #:prefix license:)
             (gnu packages autotools)
             (gnu packages base)
             (gnu packages bash)
             (gnu packages bdw-gc)
             (gnu packages compression)
             (gnu packages flex)
             (gnu packages gdb)
             (gnu packages gettext)
             (gnu packages gperf)
             (gnu packages libffi)
             (gnu packages libunistring)
             (gnu packages linux)
```

[1] Este capítulo é uma adaptação de uma postagem de blog do `https://guix.gnu.org/en/blog/2023/from-development-environments-to-continuous-integrationthe-ultimate-guide-to-software-development-with-guix/` publicada em junho de 2023 no site Guix.

```
            (gnu packages pkg-config)
            (gnu packages readline)
            (gnu packages tex)
            (gnu packages texinfo)
            (gnu packages version-control))

(package
  (name "guile")
  (version "3.0.99-git")                          ;número da versão funky
  (source #f)                                     ;nenhuma fonte
  (build-system gnu-build-system)
  (native-inputs
   (append (list autoconf
                 automake
                 libtool
                 gnu-gettext
                 flex
                 texinfo
                 texlive-base                ;para "make pdf"
                 texlive-epsf
                 gperf
                 git
                 gdb
                 strace
                 readline
                 lzip
                 pkg-config)

           ;; Ao compilar cruzadamente, uma versão nativa do próprio Guile é
           ;; needed.
           (if (%current-target-system)
               (list this-package)
               '())))
  (inputs
   (list libffi bash-minimal))
  (propagated-inputs
   (list libunistring libgc))

  (native-search-paths
   (list (search-path-specification
          (variable "GUILE_LOAD_PATH")
          (files '("share/guile/site/3.0")))
         (search-path-specification
          (variable "GUILE_LOAD_COMPILED_PATH")
          (files '("lib/guile/3.0/site-ccache")))))
  (synopsis "Scheme implementation intended especially for extensions")
  (description
```

```
   "Guile is the GNU Ubiquitous Intelligent Language for Extensions,
and it's actually a full-blown Scheme implementation!")
  (home-page "https://www.gnu.org/software/guile/")
  (license license:lgpl3+))
```

um pouco de clichê, mas agora alguém que queira hackear o Guile só precisa executar:

```
guix shell
```

Isso lhes dá um shell contendo todas as dependências do Guile: aquelas listadas acima, mas também *dependências implícitas*, como a cadeia de ferramentas GCC, GNU Make, sed, grep e assim por diante. Veja Seção "Invocando guix shell" em *Manual de Referência do GNU Guix*, para mais informações sobre `guix shell`.

Recomendação do chef: Nossa sugestão é criar ambientes de desenvolvimento como este:

```
guix shell --container --link-profile
```

... ou, para abreviar:

```
guix shell -CP
```

Isso dá um shell em um contêiner isolado, e todas as dependências aparecem em `$HOME/.guix-profile`, que funciona bem com caches como `config.cache` (veja Seção "Cache Files" em *Autoconf*) e nomes de arquivos absolutos registrados em `Makefile`s gerados e similares. O fato de o shell rodar em um contêiner traz paz de espírito: nada além do diretório atual e das dependências do Guile é visível dentro do contêiner; nada do sistema pode interferir no seu desenvolvimento.

7.2 Nível 1: Construindo com Guix

Agora que temos uma definição de pacote (veja Seção 7.1 [Começando], Página 67), por que não tirar vantagem dela também para que possamos construir o Guile com o Guix? Deixamos o campo `source` vazio, porque o `guix shell` acima só se importa com as *inputs* do nosso pacote—para que ele possa configurar o ambiente de desenvolvimento—não com o pacote em si.

Para construir o pacote com Guix, precisaremos preencher o campo `source`, desta forma:

```
(use-modules (guix)
             (guix git-download)  ;para ‘git-predicate’
             ...)

(define vcs-file?
  ;; Retorna verdadeiro se o arquivo fornecido estiver sob controle de versão.
  (or (git-predicate (current-source-directory))
      (const #t)))                                  ;não em um checkout do Git

(package
  (name "guile")
  (version "3.0.99-git")                            ;número da versão funky
  (source (local-file "." "guile-checkout"
                      #:recursive? #t
```

```
                          #:select? vcs-file?))
    ...)
```

Aqui está o que mudamos em comparação à seção anterior:

1. Adicionamos `(guix git-download)` ao nosso conjunto de módulos importados, para que possamos usar seu procedimento `git-predicate`.
2. Definimos `vcs-file?` como um procedimento que retorna true quando passado um arquivo que está sob controle de versão. Para uma boa medida, adicionamos um caso de fallback para quando não estamos em um checkout do Git: sempre retorna true.
3. Definimos `source` como um `local-file` (`https://guix.gnu.org/manual/devel/pt-br/html_node/Expressoes_002dG.html#index-local_002dfile`)—uma cópia recursiva do diretório atual (`"."`), limitado a arquivos sob controle de versão (o bit `#:select?`).

A partir daí, nosso arquivo `guix.scm` serve a um segundo propósito: ele nos permite construir o software com Guix. O ponto principal de construir com Guix é que é uma construção "limpa" — você pode ter certeza de que nada da sua árvore de trabalho ou sistema interfere no resultado da construção — e ele permite que você teste uma variedade de coisas. Primeiro, você pode fazer uma construção nativa simples:

```
guix build -f guix.scm
```

Mas você também pode compilar para outro sistema (possivelmente após configurar veja Seção "Daemon Offload Setup" em *GNU Guix Reference Manual* ou veja Seção "Serviços de virtualização" em *Manual de referência do GNU Guix*):

```
guix build -f guix.scm -s aarch64-linux -s riscv64-linux
```

. . . ou compilação cruzada:

```
guix build -f guix.scm --target=x86_64-w64-mingw32
```

Você também pode usar *transformações de pacotes* para testar variantes de pacotes (veja Seção "Opções de transformação de pacote" em *Manual de referência do GNU Guix*):

```
# E se construíssemos com Clang em vez de GCC?
guix build -f guix.scm \
--with-c-toolchain=guile@3.0.99-git=clang-toolchain

# E quanto ao sinalizador configure pouco testado?
guix build -f guix.scm \
  --with-configure-flag=guile@3.0.99-git=--disable-networking
```

Útil!

7.3 Nível 2: O Repositório como Canal

Agora temos um repositório Git contendo (entre outras coisas) uma definição de pacote (veja Seção 7.2 [Construindo com Guix], Página 69). Não podemos transformá-lo em um *channel* (veja Seção "Canais" em *Manual de referência do GNU Guix*)? Afinal, os canais são projetados para enviar definições de pacotes aos usuários, e é exatamente isso que estamos fazendo com nosso `guix.scm`.

Acontece que podemos realmente transformá-lo em um canal, mas com uma ressalva: precisamos criar um diretório separado para o(s) arquivo(s) `.scm` do nosso canal para que

`guix pull` não carregue arquivos `.scm` não relacionados quando alguém puxar o canal — e no Guile, há muitos deles! Então, começaremos assim, mantendo um link simbólico `guix.scm` de nível superior para o bem do `guix shell`:

```
mkdir -p .guix/modules
mv guix.scm .guix/modules/guile-package.scm
ln -s .guix/modules/guile-package.scm guix.scm
```

Para torná-lo utilizável como parte de um canal, precisamos transformar nosso arquivo `guix.scm` em um *módulo de pacote* (veja Seção "Módulos de pacote" em *Manual de referência do GNU Guix*): fazemos isso alterando o formulário `use-modules` no topo para um formulário `define-module`. Também precisamos realmente *exportar* uma variável de pacote, com `define-public`, enquanto ainda retornamos o valor do pacote no final do arquivo para que ainda possamos usar `guix shell` e `guix build -f guix.scm`. O resultado final se parece com isso (não repetindo coisas que não mudaram):

```
(define-module (guile-package)
  #:use-module (guix)
  #:use-module (guix git-download)   ;para 'git-predicate'
  ...)

(define vcs-file?
  ;; Retorna verdadeiro se o arquivo fornecido estiver sob controle de versão.
  (or (git-predicate (dirname (dirname (current-source-directory))))
      (const #t)))                                  ;não em um checkout do Git

(define-public guile
  (package
    (name "guile")
    (version "3.0.99-git")                          ;número da versão funky
    (source (local-file "../.." "guile-checkout"
                        #:recursive? #t
                        #:select? vcs-file?))
    ...))

;; Retorna o objeto do pacote definido acima no final do módulo.
guile
```

Precisamos de uma última coisa: um arquivo `.guix-channel` (`https://guix.gnu.org/manual/devel/pt-br/html_node/Modulos-de-pacote-em-um-subdiretorio`) para que o Guix saiba onde procurar módulos de pacote em nosso repositório:

```
;; Este arquivo nos permite apresentar este repositório como um canal Guix.

(channel
  (version 0)
  (directory ".guix/modules"))  ;procure por módulos de pacote em .guix/modules/
```

Para recapitular, agora temos estes arquivos:

```
.
 .guix-channel
```

```
guix.scm → .guix/modules/guile-package.scm
.guix
    modules
       guile-package.scm
```

E é isso: temos um canal! (Poderíamos fazer melhor e dar suporte a *autenticação de canal* (`https://guix.gnu.org/manual/devel/pt-br/html_node/Especificando-autorizacoes-de-canal.html`) para que os usuários saibam que estão extraindo código genuíno. Vamos poupá-lo dos detalhes aqui, mas vale a pena considerar!) Os usuários podem extrair deste canal por adicionando-o a `~/.config/guix/channels.scm` (`https://guix.gnu.org/manual/devel/pt-br/html_node/Especificando-canais-adicionais.html`), ao longo destas linhas:

```
(append (list (channel
               (name 'guile)
               (url "https://git.savannah.gnu.org/git/guile.git")
               (branch "main")))
        %default-channels)
```

Após executar `guix pull`, podemos ver o novo pacote:

```
$ guix describe
Geração 264 26 de maio de 2023 16:00:35 (atual)
  guile 36fd2b4
    repository URL: https://git.savannah.gnu.org/git/guile.git
    branch: main
    commit: 36fd2b4920ae926c79b936c29e739e71a6dff2bc
  guix c5bc698
    repository URL: https://git.savannah.gnu.org/git/guix.git
    commit: c5bc698e8922d78ed85989985cc2ceb034de2f23
$ guix package -A ^guile$
guile   3.0.99-git      out,debug       guile-package.scm:51:4
guile   3.0.9           out,debug       gnu/packages/guile.scm:317:2
guile   2.2.7           out,debug       gnu/packages/guile.scm:258:2
guile   2.2.4           out,debug       gnu/packages/guile.scm:304:2
guile   2.0.14          out,debug       gnu/packages/guile.scm:148:2
guile   1.8.8           out             gnu/packages/guile.scm:77:2
$ guix build guile@3.0.99-git
[...]
/gnu/store/axnzbl89yz7ld78bmx72vpqp802dwsar-guile-3.0.99-git-debug
/gnu/store/r34gsij7f0glg2fbakcmmk0zn4v62s5w-guile-3.0.99-git
```

É assim que, como desenvolvedor, você tem seu software entregue diretamente nas mãos dos usuários! Sem intermediários, mas sem perda de transparência e rastreamento de procedência.

Com isso em prática, também se torna trivial para qualquer um criar imagens Docker, pacotes Deb/RPM ou um tarball simples com `guix pack` (veja Seção “Invocando guix pack” em *Manual de Referência GNU Guix*):

```
# Que tal uma imagem Docker do nosso instantâneo Guile?
guix pack -f docker -S /bin=bin guile@3.0.99-git
```

```
# E um RPM relocável?
guix pack -f rpm -R -S /bin=bin guile@3.0.99-git
```

7.4 Bônus: Variantes do pacote

Agora temos um canal real, mas ele contém apenas um pacote (veja Seção 7.3 [O Repositório como um Canal], Página 70). Enquanto estamos nisso, podemos definir *variantes de pacote* (veja Seção "Definindo variantes de pacote" em *Manual de Referência do GNU Guix*) em nosso arquivo `guile-package.scm`, variantes que queremos poder testar como desenvolvedores Guile—semelhante ao que fizemos acima com opções de transformação. Podemos adicioná-las assim:

```
;; Este é o arquivo '.guix/modules/guile-package.scm'.

(define-module (guile-package)
  ...)

(define-public guile
  ...)

(define (package-with-configure-flags p flags)
  "Retorna P com FLAGS como sinalizadores 'configure' adicionais."
  (package/inherit p
    (arguments
     (substitute-keyword-arguments (package-arguments p)
       ((#:configure-flags original-flags #~(list))
        #~(append #$original-flags #$flags))))))

(define-public guile-without-threads
  (package
    (inherit (package-with-configure-flags guile
                                           #~(list "--without-threads")))
    (name "guile-without-threads")))

(define-public guile-without-networking
  (package
    (inherit (package-with-configure-flags guile
                                           #~(list "--disable-networking")))
    (name "guile-without-networking")))


;; Retorna o objeto do pacote definido acima no final do módulo.
guile
```

Podemos construir essas variantes como pacotes regulares assim que tivermos puxado o canal. Alternativamente, a partir de um checkout do Guile, podemos executar um comando como este do nível superior:

```
guix build -L $PWD/.guix/modules guile-without-threads
```

7.5 Nível 3: Configurando a integração contínua

O canal que definimos acima (veja Seção 7.3 [O Repositório como um Canal], Página 70) se torna ainda mais interessante quando configuramos *continuous integration* (`https://pt.wikipedia.org/wiki/Integra%C3%A7%C3%A3o_cont%C3%ADnua`) (CI). Há várias maneiras de fazer isso.

Você pode usar uma das principais ferramentas de integração contínua, como o GitLab-CI. Para fazer isso, você precisa ter certeza de executar jobs em uma imagem Docker ou máquina virtual que tenha o Guix instalado. Se fizéssemos isso no caso do Guile, teríamos um job que executa um comando shell como este:

```
guix build -L $PWD/.guix/modules guile@3.0.99-git
```

Fazer isso funciona muito bem e tem a vantagem de ser fácil de fazer na sua plataforma de CI favorita.

Dito isso, você realmente aproveitará ao máximo usando Cuirass (`https://guix.gnu.org/pt-BR/cuirass`), uma ferramenta de CI projetada para e fortemente integrada com Guix. Usá-la dá mais trabalho do que usar uma ferramenta de CI hospedada porque primeiro você precisa configurá-la, mas essa fase de configuração é bastante simplificada se você usar seu serviço Guix System (veja Seção "Integração Contínua" em *Manual de Referência GNU Guix*). Voltando ao nosso exemplo, damos ao Cuirass um arquivo de especificação que é assim:

```
;; Arquivo de especificações do Cuirass para construir todos os pacotes do canal ‘guil
(list (specification
        (name "guile")
        (build '(channels guile))
        (channels
         (append (list (channel
                        (name 'guile)
                        (url "https://git.savannah.gnu.org/git/guile.git")
                        (branch "main")))
                 %default-channels))))
```

Ela difere do que você faria com outras ferramentas de CI em dois aspectos importantes:

- O Cuirass sabe que está rastreando *dois* canais, `guile` e `guix`. De fato, nosso próprio pacote `guile` depende de muitos pacotes fornecidos pelo canal `guix`—GCC, o GNU libc, libffi e assim por diante. Mudanças em pacotes do canal `guix` podem potencialmente influenciar nossa construção `guile` e isso é algo que gostaríamos de ver o mais rápido possível como desenvolvedores do Guile.
- Os resultados da compilação não são descartados: eles podem ser distribuídos como *substitutos* para que os usuários do nosso canal `guile` obtenham binários pré-compilados de forma transparente! (veja Seção "Substitutos" em *Manual de Referência do GNU Guix*, para informações básicas sobre substitutos.)

Do ponto de vista de um desenvolvedor, o resultado final é esta status page (`https://ci.guix.gnu.org/jobset/guile`) listando *avaliações*: cada avaliação é uma combinação de confirmações dos canais `guix` e `guile`, fornecendo um número de *trabalhos* — um trabalho por pacote definido em `guile-package.scm` vezes o número de arquiteturas de destino.

Quanto aos substitutos, eles vêm de graça! Por exemplo, já que nosso jobset `guile` é construído em ci.guix.gnu.org, que executa `guix publish` (veja Seção "Invocando guix publish" em *Manual de Referência do GNU Guix*) além do Cuirass, obtém-se automaticamente substitutos para compilações `guile` de ci.guix.gnu.org; nenhum trabalho adicional é necessário para isso.

7.6 Bônus: Construir manifesto

A especificação Cuirass acima é conveniente: ela constrói todos os pacotes em nosso canal, o que inclui algumas variantes (veja Seção 7.5 [Configurando Integração Contínua], Página 74). No entanto, isso pode ser insuficientemente expressivo em alguns casos: pode-se querer trabalhos específicos de compilação cruzada, transformações, imagens Docker, pacotes RPM/Deb ou até mesmo testes de sistema.

Para conseguir isso, você pode escrever um *manifest* (veja Seção "Escrevendo manifestos" em *Manual de Referência do GNU Guix*). O que temos para Guile tem entradas para as variantes de pacote que definimos acima, bem como variantes adicionais e compilações cruzadas:

```
;; Este é '.guix/manifest.scm'.

(use-modules (guix)
             (guix profiles)
             (guile-package)) ;importa nosso próprio módulo de pacote

(define* (package->manifest-entry* package system
                                   #:key target)
  "Retorna uma entrada de manifesto para pacote PACKAGE no sisteme SYSTEM,
opcionalmente compilado cruzado para alvo TARGET."
  (manifest-entry
    (inherit (package->manifest-entry package))
    (name (string-append (package-name package) "." system
                         (if target
                             (string-append "." target)
                             "")))
    (item (with-parameters ((%current-system system)
                            (%current-target-system target))
            package))))

(define native-builds
  (manifest
   (append (map (lambda (system)
                  (package->manifest-entry* guile system))

                '("x86_64-linux" "i686-linux"
                  "aarch64-linux" "armhf-linux"
                  "powerpc64le-linux"))
           (map (lambda (guile)
                  (package->manifest-entry* guile "x86_64-linux"))
```

```
                  (cons (package
                          (inherit (package-with-c-toolchain
                                    guile
                                    `(("clang-toolchain"
                                       ,(specification->package
                                         "clang-toolchain")))))
                          (name "guile-clang"))
                        (list guile-without-threads
                              guile-without-networking
                              guile-debug
                              guile-strict-typing))))))

(define cross-builds
  (manifest
   (map (lambda (target)
          (package->manifest-entry* guile "x86_64-linux"
                                    #:target target))
        '("i586-pc-gnu"
          "aarch64-linux-gnu"
          "riscv64-linux-gnu"
          "i686-w64-mingw32"
          "x86_64-linux-gnu"))))

(concatenate-manifests (list native-builds cross-builds))
```

Não entraremos em detalhes deste manifesto; basta dizer que ele fornece flexibilidade adicional. Agora precisamos dizer ao Cuirass para construir este manifesto, que é feito com uma especificação ligeiramente diferente da anterior:

```
;; Cuirass spec file to build all the packages of the ‘guile’ channel.
(list (specification
        (name "guile")
        (build '(manifest ".guix/manifest.scm"))
        (channels
         (append (list (channel
                        (name 'guile)
                        (url "https://git.savannah.gnu.org/git/guile.git")
                        (branch "main")))
                 %default-channels))))
```

Alteramos a parte `(build ...)` da especificação para `'(manifest ".guix/manifest.scm")` para que ele escolhesse nosso manifesto, e pronto!

7.7 Empacotando

Escolhemos Guile como exemplo prático neste capítulo e você pode ver o resultado aqui:

- `.guix-channel` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix-channel?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`);

- `.guix/modules/guile-package.scm` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix/modules/guile-package.scm?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`) com o link simbólico de nível superior `guix.scm`;
- `.guix/manifest.scm` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix/manifest.scm?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`).

Hoje em dia, os repositórios são comumente recheados com arquivos dot para várias ferramentas: `.envrc`, `.gitlab-ci.yml`, `.github/workflows`, `Dockerfile`, `.buildpacks`, `Aptfile`, `requirements.txt` e outros. Pode parecer que estamos propondo um monte de arquivos *adicionais*, mas na verdade esses arquivos são expressivos o suficiente para *substituir* a maioria ou todos os listados acima.

Com alguns arquivos, obtemos suporte para:

- ambientes de desenvolvimento (`guix shell`);
- compilações de teste originais, incluindo para variantes de pacote e para compilação cruzada (`guix build`);
- integração contínua (com Cuirass ou com alguma outra ferramenta);
- entrega contínua aos usuários (*via* o canal e com binários pré-construídos);
- geração de artefatos de construção derivados, como imagens Docker ou pacotes Deb/RPM (`guix pack`).

Este é um bom (na nossa opinião!) conjunto de ferramentas unificadas para implantação de software reproduzível e uma ilustração de como você, como desenvolvedor, pode se beneficiar dele!

8 Gerenciamento de ambientes

Guix fornece múltiplas ferramentas para gerenciar o ambiente. Este capítulo demonstra tais utilitários.

8.1 Ambiente Guix via direnv

Guix fornece um pacote '`direnv`', que pode estender o shell após a mudança de diretório. Esta ferramenta pode ser usada para preparar um ambiente Guix puro.

O exemplo a seguir fornece uma função de shell para o arquivo `~/.direnvrc`, que pode ser usada no repositório Guix Git no arquivo `~/src/guix/.envrc` para configurar um ambiente de compilação semelhante ao descrito em veja Seção "Compilando do git" em *Manual de Referência do GNU Guix*.

Crie um `~/.direnvrc` com um código Bash:

```
# Obrigado <https://github.com/direnv/direnv/issues/73#issuecomment-152284914>
export_function()
{
  local name=$1
  local alias_dir=$PWD/.direnv/aliases
  mkdir -p "$alias_dir"
  PATH_add "$alias_dir"
  local target="$alias_dir/$name"
  if declare -f "$name" >/dev/null; then
    echo "#!$SHELL" > "$target"
    declare -f "$name" >> "$target" 2>/dev/null
    # Observe que adicionamos variáveis de shell ao gatilho da função.
    echo "$name \$*" >> "$target"
    chmod +x "$target"
  fi
}

use_guix()
{
    # Defina o token do GitHub.
    export GUIX_GITHUB_TOKEN="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

    # Unset 'GUIX_PACKAGE_PATH'.
    export GUIX_PACKAGE_PATH=""

    # Recrie uma raiz do coletor de lixo.
    gcroots="$HOME/.config/guix/gcroots"
    mkdir -p "$gcroots"
    gcroot="$gcroots/guix"
    if [ -L "$gcroot" ]
    then
        rm -v "$gcroot"
```

```
fi

# Pacotes diversos.
PACKAGES_MAINTENANCE=(
    direnv
    git
    git:send-email
    git-cal
    gnupg
    guile-colorized
    guile-readline
    less
    ncurses
    openssh
    xdot
)

# Pacotes de ambiente.
PACKAGES=(help2man guile-sqlite3 guile-gcrypt)

# Obrigado <https://lists.gnu.org/archive/html/guix-devel/2016-09/msg00859.html>
eval "$(guix shell --search-paths --root="$gcroot" --pure \
 --development guix ${PACKAGES[@]} ${PACKAGES_MAINTENANCE[@]} "$@")"

# Predefina sinalizadores de configuração.
configure()
{
    ./configure
}
export_function configure

# Execute make e, opcionalmente, construa algo.
build()
{
    make -j 2
    if [ $# -gt 0 ]
    then
        ./pre-inst-env guix build "$@"
    fi
}
export_function build

# Predefina o comando push do Git.
push()
{
    git push --set-upstream origin
}
```

```
    export_function push

    clear                         # Limpe a tela.
    git-cal --author='Seu nome' # Mostrar calendário de contribuições.

    #Mostrar ajuda de comandos.
    echo "
build          construir um pacote ou apenas um projeto se nenhum argumento for fornec
configure      execute ./configure com parâmetros predefinidos
push           enviar para o repositório Git upstream
"
}
```

Cada projeto contendo `.envrc` com uma string `use guix` terá variáveis de ambiente e procedimentos predefinidos.

Execute `direnv allow` para configurar o ambiente pela primeira vez.

9 Instalando Guix em um Cluster

O Guix é atraente para cientistas e profissionais de HPC (computação de alto desempenho): ele facilita a implantação de pilhas de software potencialmente complexas e permite que você faça isso de forma reproduzível — você pode reimplantar exatamente o mesmo software em máquinas diferentes e em momentos diferentes.

Neste capítulo, veremos como um administrador de sistema de cluster pode instalar o Guix para uso em todo o sistema, de modo que ele possa ser usado em todos os nós do cluster, e discutiremos as várias compensações[1].

> **Nota:** Aqui, assumimos que o cluster está executando uma distribuição GNU/Linux diferente do Guix System e que vamos instalar o Guix sobre ele.

9.1 Configurando um nó principal

A abordagem recomendada é configurar um *nó principal* executando `guix-daemon` e exportando `/gnu/store` via NFS para nós de computação.

Lembre-se de que `guix-daemon` é responsável por gerar processos de compilação e downloads em nome dos clientes (veja Seção "Invocando guix-daemon" em *Manual de Referência do GNU Guix*) e, de forma mais geral, acessando `/gnu/store`, que contém todos os binários de pacotes compilados por todos os usuários (veja Seção "O armazém" em *Manual de Referência do GNU Guix*). "Cliente" aqui se refere a todos os comandos Guix que os usuários veem, como `guix install`. Em um cluster, esses comandos podem estar em execução nos nós de computação e queremos que eles conversem com a instância `guix-daemon` do nó principal.

Para começar, o nó principal pode ser instalado seguindo as instruções usuais de instalação binária (veja Seção "Instalação de binários" em *Manual de Referência do GNU Guix*). Graças ao script de instalação, isso deve ser rápido. Assim que a instalação for concluída, precisamos fazer alguns ajustes.

Como queremos que o `guix-daemon` seja acessível não apenas do nó principal, mas também dos nós de computação, precisamos organizar para que ele escute conexões por TCP/IP. Para fazer isso, editaremos o arquivo de inicialização do systemd para `guix-daemon`, `/etc/systemd/system/guix-daemon.service` e adicionaremos um argumento `--listen` à linha `ExecStart` para que fique parecido com isto:

```
ExecStart=/var/guix/profiles/per-user/root/current-guix/bin/guix-daemon \
    --build-users-group=guixbuild \
    --listen=/var/guix/daemon-socket/socket --listen=0.0.0.0
```

Para que essas alterações entrem em vigor, o serviço precisa ser reiniciado:

```
systemctl daemon-reload
systemctl restart guix-daemon
```

> **Nota:** O bit `--listen=0.0.0.0` significa que `guix-daemon` processará *todas* as conexões TCP de entrada na porta 44146 (veja Seção "Invocando guix-daemon"

[1] Este capítulo foi adaptado de uma postagem de blog do publicada no site Guix-HPC em 2017 (`https://hpc.guix.info/blog/2017/11/installing-guix-on-a-cluster/`).

em *Manual de Referência do GNU Guix*). Isso geralmente é bom em uma configuração de cluster onde o nó principal é acessível exclusivamente da rede local do cluster — você não quer que isso seja exposto à Internet!

O próximo passo é definir nossas exportações NFS em `/etc/exports` (`https://linux.die.net/man/5/exports`) adicionando algo como isto:

```
/gnu/store    *(ro)
/var/guix     *(rw, async)
/var/log/guix *(ro)
```

O diretório `/gnu/store` pode ser exportado somente para leitura, pois somente `guix-daemon` no nó mestre poderá modificá-lo. `/var/guix` contém *perfis de usuário* conforme gerenciados por `guix package`; portanto, para permitir que os usuários instalem pacotes com `guix package`, isso deve ser de leitura e gravação.

Os usuários podem criar quantos perfis quiserem, além do perfil padrão, `~/.guix-profile`. Por exemplo, `guix package -p ~/dev/python-dev -i python` instala o Python em um perfil acessível pelo link simbólico `~/dev/python-dev`. Para garantir que esse perfil esteja protegido contra coleta de lixo — ou seja, que o Python não será removido de `/gnu/store` enquanto esse perfil existir —, *diretórios home devem ser montados no nó principal* também para que `guix-daemon` saiba sobre esses perfis não padrão e evite coletar softwares aos quais eles se referem.

Pode ser uma boa ideia remover periodicamente bits não utilizados de `/gnu/store` executando `guix gc` (veja Seção "Invocando guix gc" em *Manual de Referência do GNU Guix*). Isso pode ser feito adicionando uma entrada crontab no nó principal:

```
root@master# crontab -e
```

... com algo assim:

```
# Todos os dias às 5h da manhã, execute o coletor de lixo para garantir
# pelo menos 10 GB estão livres em /gnu/store.
0 5 * * 1 /usr/local/bin/guix gc -F10G
```

Terminamos com o nó principal! Vamos dar uma olhada nos nós de computação agora.

9.2 Configurando nós de computação

Primeiro, precisamos de nós de computação para montar os diretórios NFS que o nó principal exporta. Isso pode ser feito adicionando as seguintes linhas a `/etc/fstab` (`https://linux.die.net/man/5/fstab`):

```
head-node:/gnu/store    /gnu/store    nfs  defaults,_netdev,vers=3 0 0
head-node:/var/guix     /var/guix     nfs  defaults,_netdev,vers=3 0 0
head-node:/var/log/guix /var/log/guix nfs  defaults,_netdev,vers=3 0 0
```

... onde *head-node* é o nome ou endereço IP do seu nó principal. A partir daí, assumindo que os pontos de montagem existam, você deve conseguir montar cada um deles nos nós de computação.

Em seguida, precisamos fornecer um comando `guix` padrão que os usuários possam executar quando se conectarem ao cluster pela primeira vez (eventualmente, eles invocarão `guix pull`, que fornecerá a eles seu "próprio" comando `guix`). Semelhante ao que o script de instalação binário fez no nó principal, armazenaremos isso em `/usr/local/bin`:

```
mkdir -p /usr/local/bin
```

```
ln -s /var/guix/profiles/per-user/root/current-guix/bin/guix \
      /usr/local/bin/guix
```

Precisamos então dizer ao `guix` para falar com o daemon em execução no nosso nó mestre, adicionando estas linhas ao `/etc/profile`:

```
GUIX_DAEMON_SOCKET="guix://head-node"
export GUIX_DAEMON_SOCKET
```

Para evitar avisos e garantir que o `guix` use o local correto, precisamos instruí-lo a usar os dados de local fornecidos pelo Guix (veja Seção "Configuração de aplicativo" em *Manual de Referência do GNU Guix*):

```
GUIX_LOCPATH=/var/guix/profiles/per-user/root/guix-profile/lib/locale
export GUIX_LOCPATH

# Aqui, devemos usar um nome de localidade válido. Tente "ls $GUIX_LOCPATH/*"
# para ver quais nomes podem ser usados.
LC_ALL=fr_FR.utf8
export LC_ALL
```

Por conveniência, `guix package` gera automaticamente `~/.guix-profile/etc/profile`, que define todas as variáveis de ambiente necessárias para usar os pacotes—`PATH`, `C_INCLUDE_PATH`, `PYTHONPATH`, etc. Da mesma forma, `guix pull` faz isso em `~/.config/guix/current`. Portanto, é uma boa ideia obter ambos de `/etc/profile`:

```
for GUIX_PROFILE in "$HOME/.config/guix/current" "$HOME/.guix-profile"
do
  if [ -f "$GUIX_PROFILE/etc/profile" ]; then
    . "$GUIX_PROFILE/etc/profile"
  fi
done
```

Por último, mas não menos importante, o Guix fornece conclusão de linha de comando, notavelmente para Bash e zsh. Em `/etc/bashrc`, considere adicionar esta linha:

```
. /var/guix/profiles/per-user/root/current-guix/etc/bash_completion.d/guix
```

Voilà!

Você pode verificar se tudo está no lugar efetuando login em um nó de computação e executando:

```
guix install hello
```

O daemon no nó principal deve baixar binários pré-compilados em seu nome e descompactá-los em `/gnu/store`, e `guix install` deve criar `~/.guix-profile` contendo o comando `~/.guix-profile/bin/hello`.

9.3 Acesso à rede

Guix requer acesso à rede para baixar código-fonte e binários pré-construídos. A boa notícia é que somente o nó principal precisa disso, já que nós de computação simplesmente delegam a ele.

É costume que nós de cluster tenham acesso, na melhor das hipóteses, a uma *white list* de hosts. Nosso nó principal precisa de pelo menos `ci.guix.gnu.org` nessa lista branca,

pois é de lá que ele obtém binários pré-construídos por padrão, para todos os pacotes que estão no Guix propriamente dito.

A propósito, `ci.guix.gnu.org` também serve como um *content-addressed mirror* do código-fonte desses pacotes. Consequentemente, é suficiente ter *unicamente* `ci.guix.gnu.org` nessa lista branca.

Pacotes de software mantidos em um repositório separado, como um dos vários canais HPC (`https://hpc.guix.info/channels`), obviamente não estão disponíveis em `ci.guix.gnu.org`. Para esses pacotes, você pode querer estender a lista branca de modo que os binários de origem e pré-construídos (assumindo que os servidores desta parte forneçam binários para esses pacotes) possam ser baixados. Como último recurso, os usuários sempre podem baixar o código-fonte em sua estação de trabalho e adicioná-lo ao `/gnu/store` do cluster, assim:

```
GUIX_DAEMON_SOCKET=ssh://compute-node.example.org \
  guix download http://starpu.gforge.inria.fr/files/starpu-1.2.3/starpu-1.2.3.tar.gz
```

O comando acima baixa `starpu-1.2.3.tar.gz` *e* o envia para a instância `guix-daemon` do cluster via SSH.

Clusters com air-gapped exigem mais trabalho. No momento, nossa sugestão seria baixar todo o código-fonte necessário em uma estação de trabalho executando Guix. Por exemplo, usando a opção `--sources` de `guix build` (veja Seção "Invocando guix build" em *Manual de Referência do GNU Guix*), o exemplo abaixo baixa todo o código-fonte do qual o pacote `openmpi` depende:

```
$ guix build --sources=transitive openmpi

...

/gnu/store/xc17sm60fb8nxadc4qy0c7rqph499z8s-openmpi-1.10.7.tar.bz2
/gnu/store/s67jx92lpipy2nfj5cz818xv430n4b7w-gcc-5.4.0.tar.xz
/gnu/store/npw9qh8a46lrxiwh9xwk0wpi3jlzmjnh-gmp-6.0.0a.tar.xz
/gnu/store/hcz0f4wkdbsvsdky3c0vdvcawhdkyldb-mpfr-3.1.5.tar.xz
/gnu/store/y9akh452n3p4w2v631nj0injx7y0d68x-mpc-1.0.3.tar.gz
/gnu/store/6g5c35q8avfnzs3v14dzl54cmrvddjm2-glibc-2.25.tar.xz
/gnu/store/p9k48dk3dvvk7gads7fk30xc2pxsd66z-hwloc-1.11.8.tar.bz2
/gnu/store/cry9lqidwfrfmgl0x389cs3syr15p13q-gcc-5.4.0.tar.xz
/gnu/store/7ak0v3rzpqm2c5q1mp3v7cj0rxz0qakf-libfabric-1.4.1.tar.bz2
/gnu/store/vh8syjrsilnbfcf582qhmvpg1v3rampf-rdma-core-14.tar.gz
...
```

(Caso você esteja se perguntando, isso é mais de 320 MiB de código-fonte *compactado*.)

Podemos então criar um grande arquivo contendo tudo isso (veja Seção "Invocando guix archive" em *Manual de Referência GNU Guix*):

```
$ guix archive --export \
    `guix build --sources=transitive openmpi` \
    > openmpi-source-code.nar
```

... e podemos eventualmente transferir esse arquivo para o cluster em armazenamento removível e descompactá-lo lá:

```
$ guix archive --import < openmpi-source-code.nar
```

Esse processo precisa ser repetido toda vez que um novo código-fonte precisa ser trazido para o cluster.

Enquanto escrevemos isso, os institutos de pesquisa envolvidos no Guix-HPC não têm clusters air-gapped. Se você tem experiência com essas configurações, gostaríamos de ouvir feedback e sugestões.

9.4 Uso do Disco

Uma preocupação comum dos administradores de sistemas é se tudo isso vai consumir muito espaço em disco. Se alguma coisa vai esgotar o espaço em disco, serão conjuntos de dados científicos em vez de software compilado — essa é nossa experiência com quase dez anos de uso do Guix em clusters HPC. No entanto, vale a pena dar uma olhada em como o Guix contribui para o uso do disco.

Primeiro, ter várias versões ou variantes de um determinado pacote em `/gnu/store` não custa necessariamente muito, porque `guix-daemon` implementa a desduplicação de arquivos idênticos, e as variantes de pacote provavelmente terão vários arquivos em comum.

Conforme mencionado acima, recomendamos ter um cron job para executar `guix gc` periodicamente, o que remove o software *não utilizado* de `/gnu/store`. No entanto, sempre há a possibilidade de que os usuários mantenham muitos softwares em seus perfis, ou muitas gerações antigas de seus perfis, que são "ativos" e não podem ser excluídos do ponto de vista do `guix gc`.

A solução para isso é que os usuários removam regularmente gerações antigas de seus perfis. Por exemplo, o comando a seguir remove gerações com mais de dois meses de idade:

```
guix package --delete-generations=2m
```

Da mesma forma, é uma boa ideia convidar os usuários a atualizarem regularmente seus perfis, o que pode reduzir o número de variantes de um determinado software armazenado em `/gnu/store`:

```
guix pull
guix upgrade
```

Como último recurso, é sempre possível para os administradores de sistemas fazerem parte disso em nome de seus usuários. No entanto, um dos pontos fortes do Guix é a liberdade e o controle que os usuários têm sobre seu ambiente de software, então recomendamos fortemente deixar os usuários no controle.

9.5 Considerações de segurança

Em um cluster HPC, o Guix é normalmente usado para gerenciar software científico. Softwares críticos de segurança, como o kernel do sistema operacional e serviços do sistema, como `sshd` e o agendador de lotes, permanecem sob o controle dos administradores de sistema.

O projeto Guix tem um bom histórico de entrega de atualizações de segurança em tempo hábil (veja Seção "Atualizações de segurança" em *Manual de Referência do GNU Guix*). Para obter atualizações de segurança, os usuários precisam executar `guix pull && guix upgrade`.

Como o Guix identifica exclusivamente variantes de software, é fácil ver se um pedaço vulnerável de software está em uso. Por exemplo, para verificar se a variante glibc 2.25

sem o patch de mitigação contra “Stack Clash (`https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt`)”, pode-se verificar se os perfis de usuário fazem referência a ela:

```
guix gc --referrers /gnu/store/...-glibc-2.25
```

Isso informará se existem perfis que se referem a essa variante glibc específica.

10 Agradecimentos

Guix é baseado no gerenciador de pacotes Nix (`https://nixos.org/nix/`), que foi projetado e implementado por Eelco Dolstra, com contribuições de outras pessoas (veja o arquivo `nix/AUTHORS` no Guix.) O Nix foi pioneiro no gerenciamento funcional de pacotes e promoveu recursos sem precedentes, como atualizações e reversões de pacotes transacionais, perfis por usuário e processos de construção referencialmente transparentes. Sem esse trabalho, o Guix não existiria.

As distribuições de software baseadas em Nix, Nixpkgs e NixOS, também foram uma inspiração para o Guix.

O GNU Guix em si é um trabalho coletivo com contribuições de várias pessoas. Veja o arquivo `AUTHORS` no Guix para obter mais informações sobre essas pessoas legais. O arquivo `THANKS` lista as pessoas que ajudaram a relatar erros, cuidar da infraestrutura, fornecer ilustrações e temas, fazer sugestões e muito mais – obrigado!

Este documento inclui seções adaptadas de artigos que foram publicados anteriormente no blog Guix em `https://guix.gnu.org/blog` e no blog Guix-HPC em `https://hpc.guix.info/blog`.

Apêndice A Licença de Documentação Livre GNU

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Índice de conceitos