

IT 資格 道 場

LPIC-2 201試験 学習用テキスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

無料サンプル(第1章まで)

LPI公式 LPIC-2 Exam 201 Objectives Version 4.5 準拠

2026年10月版

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

目次

この教材群の使い方(3点セット)

学習を始める前に:この教材の使い方

第1章 システムの起動とブートローダー [202.1/202.2/202.3]

1-1 起動の四段階 [202.2]

1-2 ファームウェア:BIOS と UEFI [202.2]

1-3 GRUB 2 [202.2]

1-4 initramfs:ルートをマウントするための一時的な環境 [202.2]

1-5 init:systemd と、その前の世代 [202.1]

1-6 代替ブートローダー [202.3]

1-7 章末チェック [202.1/202.2/202.3]

サンプルはここまでです

この教材群の使い方(3点セット)

種類	役割
① 学習用テキスト	これだけで問題が解けるようになる本編
② 頻出度別ノート	テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡し。試験直前の最終チェックにも
③ 暗記用ノート	覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

このテキストの位置づけ: ①の学習用テキストです。まずここを通読し、これだけで問題が解ける状態を目指してください。

学習を始める前に:この教材の使い方

このテキストの目的は、**LPIC-2 201試験(試験コード 201-450)** の合格です(実施は Linux Professional Institute. 略して LPI)。

LPIC-1 は「動いている Linux を操作・運用する」知識を問います。201試験が問うのは、**Linux サーバーを組み立て、動きを測り、壊れたときに切り分ける側**の知識です。そのため、次の3点まで踏み込みます。

1. **似ているコマンドの役割の違い**(`lvextend` と `resize2fs`、`dump` と `dumpe2fs` など)

2. **手順の順番**(`pvccreate` → `vgcreate` → `lvcreate` を飛ばせない理由など)

3. 設定が「今だけ」か「次回起動後も」か (`sysctl -w` と `/etc/sysctl.conf` など)

以降はこの 3 点に紙面を集中させ、man ページの網羅はしません。

試験の基本情報

項目	内容
試験名	LPIC-2 201試験 (201-450)
運営	Linux Professional Institute (LPI)
出題範囲のバージョン	Version 4.5.0 (2017-02-13 発効)
認定要件	201試験と202試験の両方に合格すると LPIC-2 に認定される。認定の付与には有効な LPIC-1 認定が必要
問題数	60問
出題形式	択一 (multiple-choice) と記述入力 (fill-in-the-blank)
試験時間	90分
合格ライン	200~800 のスケールで 500点。必要な正答数は出題の組み合わせによって変わる
受験料 (日本)	18,000円 (201試験・2026-09 時点。税込／税抜の別は公式に記載なし)
試験言語	日本語・英語ほか
受験方法	ピアソン VUE のテストセンター、または自宅等でのオンライン受験 (OnVUE)
認定の有効期間	5年

出題比重は重み (weight) で示され、合計が 60 です。重み 1 につき本番で 1 問が出ます (各章の冒頭に副目標と重みを書いてあるので、時間の配分に使える)。受験料や仕様は改定されることがあるので、申し込み前に公式サイトで確かめてください。

構成 (全 9 章)

公式の出題範囲 (Version 4.5.0) の Topic 200～206 を、副目標を割らずに 8 章へまとめています。第9章は、第1～8章で扱いきれなかった用語の分野別の補いです。

章	扱う内容	担当する副目標 (重み)
第1章	システムの起動とブートローダー	202.1(3)・202.2(4)・202.3(2)
第2章	Linux カーネル	201.1(2)・201.2(3)・201.3(4)
第3章	ファイルシステム	203.1(4)・203.2(3)・203.3(2)
第4章	RAID・ストレージデバイス・LVM	204.1(3)・204.2(2)・204.3(3)
第5章	ネットワーク構成	205.1(3)・205.2(4)
第6章	ネットワークの問題解決	205.3(4)
第7章	リソース測定と需要予測	200.1(6)・200.2(2)
第8章	ビルド・バックアップ・通知	206.1(2)・206.2(3)・206.3(1)
第9章	用語集 (分野別)	全副目標の補い

読み方:3 つの約束

1. ##### で始まる「書式ブロック」が、そのコマンドの正典（書式・オプション表・例 行）。別の章では節番号を示し、その章に固有の使い方だけを置く
2. 例 の行は、記述入力 (fill-in-the-blank) の答えになりうる形。長形と短形のように同じ意味の書き方は、本文で「同じ」と書く
3. 各章の末尾に「混同しやすい対の概念」「この章の必修コマンド一覧」「試験ポイント」。読み終えた直後と試験直前の見返しに使う

第1章 システムの起動とブートローダー

[202.1/202.2/202.3]

この章は公式副目標 **202.1 システム起動のカスタマイズ**・**202.2 システムのリカバリ**・**202.3 代替ブートローダー** に当たります。

電源を入れてからログインプロンプトが出るまで、Linux は**性質の違う四つの段階**を通ります(ファームウェア → ブートローダー → initramfs → init)。段階ごとに、動いている場所も使える道具も違います。設問の多くは「**いま止まっているのはどの段階か**」を問うので、この章も段階の順に進めます。

起動の四段階 —— 動いている場所も、使える道具も違う

設問が問うているのは「いま止まっているのはどの段階か」。

①ファームウェア	➡	②ブートローダー	➡	③initramfs	➡	④init
動いているもの						
BIOS または UEFI (マザーボード上の 不揮発メモリ)		GRUB 2・LILO・ SYSLINUX 系・ systemd-boot など		RAM 上に展開された 一時的なルート ファイルシステム		systemd (あるいは SysVinit・Upstart)
この段階で使える道具						
ファームウェア設定 画面、UEFI シェル		GRUB のメニュー編集 grub> grub rescue>		initramfs 内の BusyBox シェル		systemctl systemd-analyze journalctl

Linux のユーザーランドのコマンドが使えるのは④以降。mount・chroot・systemctl は②の grub rescue> プロンプトからは実行できません。

図 1-1 起動の四段階:動いている場所も、使える道具も違う

1-1 起動の四段階 [202.2]

段階	動いているもの	主な仕事	この段階で使える道具
①ファームウェア	BIOS または UEFI (マザーボード上の 不揮発メモリ)	自己診断 (POST)、 起動デバイスの選 択、最初のコードの 読み出し	ファームウェア設定 画面、UEFI シェル
②ブートローダー	GRUB 2・LILO・ SYSLINUX 系・ systemd-boot・U- Boot など	カーネルと initramfs をメモリ へ読み込み、カーネ ルパラメータを渡し て制御を移す	GRUB のメニュー編 集 (e)、grub> / grub rescue> プ ロンプト
③initramfs	RAM 上に展開され た一時的なルートフ ァイルシステム	本来のルートファイ ルシステムをマウン トするためのモジュ ールと道具を提供 し、switch_root で切り替える	initramfs 内の BusyBox シェル
④init	systemd (あるいは SysVinit・Upstart)	ユニットを依存関係 に従って起動し、目 標とする状態 (ター ゲット／ランレベ ル) へ到達させる	systemctl ・ systemd- analyze ・ journalctl

Linux のコマンド (mount ・ chroot ・ systemctl ・ efibootmgr ・ grub2-mkconfig) が使えるのは④以降です。②の grub rescue>からは実行できません。逆に set root= ・ linux ・ initrd ・ normal は GRUB の内蔵コマンドです。Linux のシェルにはありません。

⚠ insmod だけは両方の段階にあります。GRUB の insmod は、GRUB のモジュール (ext2 ・ lvm ・ normal など) を読みます。Linux の insmod は、カーネ

ルモジュール(`.ko`)を読む別のプログラムです(2-9-4)。

BIOS の典型的な流れは次のとおりです。

電源投入

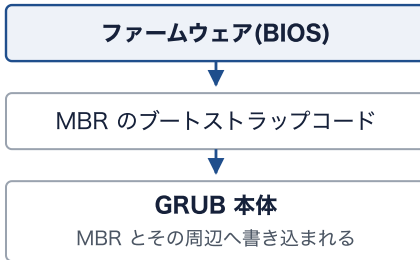
- ファームウェア (BIOS) の自己診断 (POST)
- ディスク先頭 512 バイト (MBR) のブートストラップコードの実行
- GRUB がカーネルと (必要なら) `initramfs` をメモリへ読み込む
- カーネルが起動し、`initramfs` を経て実際のルートファイルシステムへ切り替える
- `init` (`systemd`) がサービスを起動する

カーネルのロードが POST より先に来ることはありません。ルートファイルシステムをマウントするのも、GRUB ではなくカーネルです (GRUB はカーネルをメモリへ載せるところまで)。

1-2 ファームウェア: BIOS と UEFI [202.2]

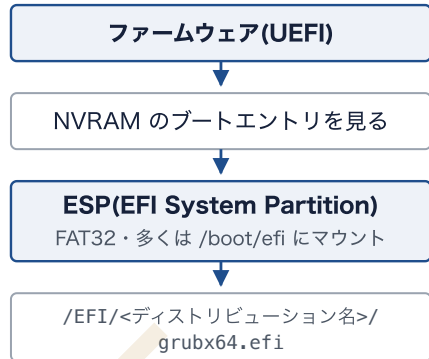
起動の入口には **BIOS レガシーブート**と **UEFI ブート**の二つがあり、どこに何を置くかの約束事が根本から違います。

BIOS/MBR 方式



GPT のディスクを BIOS レガシーブートさせるには BIOS Boot Partition が必要 (bios_grub フラグ・ファイルシステム無し)

UEFI ブート方式



ブートエントリの操作は efibootmgr

efibootmgr -o = 恒久的なブート順序(BootOrder) / efibootmgr -n = 次回 1 回限り
/boot/efi は ESP そのもの、/boot/grub2(Debian 系は /boot/grub)は grub.cfg の置き場 —— 別物

図 1-2 BIOS レガシーブートと UEFI ブート:ファームウェアが何を読みに行
くか

1-2-1 BIOS レガシーブート:ディスクの先頭を読む

BIOS は自己診断 (POST) を終わると、選ばれたディスクの**先頭 512 バイトをそのままメモリへ読み込んで実行**します。この 512 バイトが **MBR (Master Boot Record)** です。

BIOS はファイルシステムを理解せず、`/boot/vmlinuz` を探すこともできません。そのため 512 バイトの小さなコードから始め、そのコードが次の段階を呼ぶ多段構えになります。

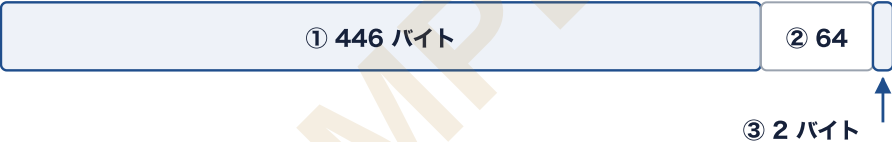
1-2-2 MBR の 512 バイトの内訳

MBR は LBA 0 (ディスクの先頭セクタ) に置かれた 512 バイトの領域で、内訳が決まっています。

部分	大きさ	中身
ブートストラップコード	446 バイト	第 1 段階のブートローダー (GRUB では <code>boot.img</code> 相当)
パーティションテーブル	64 バイト	16 バイト × 4 エントリ (基本パーティション 4 つぶん)
ブートシグネチャ	2 バイト	<code>0x55AA</code> 。この 2 バイトが無いと BIOS は起動可能と見なさない

MBR(Master Boot Record)= 512 バイト

ディスク先頭セクタ (LBA0)。内訳の足し算ごとと覚える。



① ブートストラップコード	446 バイト	最初に実行される起動用のコード
② パーティションテーブル	64 バイト	基本パーティション 4 つ分の情報
③ ブートシグネチャ	2 バイト	0x55AA。有効な MBR である印

446 + 64 + 2 = 512

この足し算ごとと覚えるのが確実です。

図 1-3 MBR 512 バイトの内訳: 446 + 64 + 2 = 512

446 + 64 + 2 = 512 はそのまま設問になります。数 MB あるカーネル (`vmlinux`) や `initramfs` はここに入らず、`/boot` のファイルシステムに置かれます。

MBR の制約は、この 64 バイト (4 エントリ) から来ます (基本パーティションは 4 つまで、1 パーティション 2 TiB まで。512 バイトセクタの場合)。

1-2-3 UEFI ブート:ファームウェアがファイルを読む

UEFI ファームウェアは**ファイルシステム (FAT) を読めます**。そこに置かれた **.efi の実行ファイル (EFI アプリケーション) を直接実行** します。そのため 512 バイトから多段で始める必要はありません。`grubx64.efi` のようなブートローダー本体をいきなり起動します (ディスク先頭の起動コードは使わない)。

どの `.efi` を起動するかは、**ファームウェアの NVRAM に「ブートエントリ」として記録** されています。`BootOrder` が試す順序を、`Boot0000`・`Boot0001` … が各エントリ (ラベルとローダーのパス) を持ちます。BIOS の「起動ドライブの順序」が、OS から操作できる変数になったものです。

1-2-4 ESP (EFI System Partition)

UEFI ファームウェアが `.efi` を読みに行く先が **ESP (EFI System Partition)** です。

- **FAT32 (vfat) でフォーマットする** (ext4 や XFS はファームウェアが読めない)
- **パーティションテーブル上に実在する通常のパーティション**で、MBR (ディスク先頭 512 バイト) とは別物
- **多くのディストリビューションで `/boot/efi` にマウントされる**。配下の `/EFI/<ディストリ名>/` に `grubx64.efi` や `shimx64.efi` が置かれる

ディレクトリ	実体	中身
<code>/boot/efi/</code>	ESP (FAT32 の別パーティション)	ブートローダーの <code>.efi</code> ファイル

ディレクトリ	実体	中身
/boot/grub2/ (Debian 系は /boot/grub/)	通常はルートファイルシステム上のただのディレクトリ	GRUB の設定ファイル grub.cfg と GRUB のモジュール類

UEFI ブートでは両方を同時に使います (ESP の grubx64.efi が起動し、その GRUB が /boot/grub2/grub.cfg を読む)。シンボリックリンクの関係でもありません。vmlinuz と initramfs は、通常 /boot/ の直下に置かれます。

1-2-5 GPT と BIOS Boot Partition

MBR のパーティションテーブルに代わる新しい方式が **GPT (GUID Partition Table)** です。GPT ではプライマリヘッダが LBA 1 に置かれます (LBA 0 は互換用の保護 MBR)。パーティション数の上限も 2 TiB の壁も実質的に無くなります。UEFI は GPT を前提としますが、**GPT と UEFI は別々の話**で、組み合わせは自由です。

GPT のディスクを BIOS レガシーブートで起動する場合を考えます。MBR 方式なら、MBR 直後の未使用領域に GRUB のコアイメージ (core.img) を書けます。GPT ではこの場所を当てにできません。そこで **BIOS Boot Partition** を用意します。

- パーティションタイプを **BIOS boot** にする (parted では bios_grub フラグ。GUID は 21686148-6449-6E6F-744E-656564454649)
- **ファイルシステムを持たない**。フォーマットしないし、マウントもしない
- 大きさは**数百 KB～1 MB 程度**で足りる

ESP は UEFI ブートで使う FAT32 のパーティションです。**BIOS Boot Partition は GPT + BIOS ブートで使うフォーマットしない小領域**で、役割が正反対です。/boot パーティション (ext4 でカーネルを置く) とも別です。

1-2-6 Secure Boot と shim

Secure Boot は、ファームウェアに登録された鍵で `.efi` の署名を検証し、通ったブートローダーしか実行しない仕組みです。Microsoft の鍵はほぼ全てのファームウェアに入っていますが、各ディストリビューションの鍵は入っていません。これを **Secure Boot を有効なまま解決**するのが、小さなブートローダーの **shim** です。

- **shim は Microsoft によって署名済み**なので、ファームウェアの検証をそのまま通る
- 起動した shim が、**shim 自身の鍵か、MOK (Machine Owner Key) として登録された鍵**で GRUB 本体を検証する。通れば起動する
- 利用者は MOK に自分の鍵を登録することで、自分でビルドしたモジュールやカーネルも通せる

検証の連鎖を一段挟んで、Secure Boot を切らずに済ませる方式です。次の説明はどれも誤りです。

- 「`efibootmgr` で Secure Boot を無効化する」：有効なまま保てない
- 「MBR のブートコードとして書き込む」：UEFI で MBR は使わない
- 「`initramfs` に GPG 鍵を埋め込む」：ファームウェアの検証とは階層が違う

1-2-7 UEFI シェル

UEFI シェルは、OS の起動前にファームウェアの段階で動くコマンドライン環境です。ファームウェアが提供するか、EFI アプリケーションとして起動します。

- ESP 上のファイルシステムを参照できる (`fs0:` のような名前でボリュームを指す)
- `.efi` ファイルを名前で指定して手動で実行できる

- ブート関連の変数を確認・操作できる

NVRAM のブートエントリが壊れて起動しないとき、ESP 上のローダーを手で実行して起動する復旧に使います。

「シェル」は階層で区別します。

シェル	動く階層	いつ出てくるか
UEFI シェル	ファームウェア (OS 起動前)	ブートエントリが失われたときなどに、ファームウェアから起動する
<code>grub></code> / <code>grub rescue></code>	ブートローダー (OS 起動前)	GRUB が設定ファイルを読めない・メニューを構築できないとき
initramfs 内の BusyBox シェル	initramfs (カーネル起動後・ルート切り替え前)	本来のルートファイルシステムのマウントに失敗したとき
<code>rescue.target</code> / <code>emergency.target</code> のシェル	init (OS 起動後)	<code>systemctl</code> で切り替えたとき、あるいはカーネルパラメータで指定したとき

UEFI シェルは Linux カーネルの機能でも systemd の機能でもありません。
`systemctl rescue` で着くのは `rescue.target` です。`grub-install` で使えるようになるものでも、ログインシェルでもありません。

1-2-8 `efibootmgr` :NVRAM のブートエントリを操作する

`efibootmgr` :UEFI ブートエントリの一覧・作成・削除・順序変更

書式 `efibootmgr` [オプション]

オプション	長形	意味
<code>-v</code>	<code>--verbose</code>	各エントリのデバイスパス・ローダーファイルのパスまで詳細に表示する
<code>-c</code>	<code>--create</code>	新しいブートエントリを作成し、 <code>BootOrder</code> の先頭へ加える
<code>-b</code>	<code>--bootnum</code>	操作対象のエントリ番号 (16 進 4 桁) を指定する
<code>-B</code>	<code>--delete-bootnum</code>	<code>-b</code> で指定したエントリを削除する
<code>-d</code>	<code>--disk</code>	ローダーが置かれたディスク (既定は <code>/dev/sda</code>)
<code>-p</code>	<code>--part</code>	ローダーが置かれたパーティション番号 (既定は 1)
<code>-l</code>	<code>--loader</code>	ローダーファイルのパス (<code>\EFI\...\grubx64.efi</code> の形)
<code>-n</code>	<code>--bootnext</code>	次回 1 回限りで使うエントリ (<code>BootNext</code>) を設定する
<code>-N</code>	<code>--delete-bootnext</code>	<code>BootNext</code> を削除する
<code>-o</code>	<code>--bootorder</code>	<code>BootOrder</code> (恒久的な起動順序) をカンマ区切りで設定する
<code>-O</code>	<code>--delete-bootorder</code>	<code>BootOrder</code> 変数そのものを削除する

オプション	長形	意味
<code>-a</code>	<code>--active</code>	指定エントリを有効 (active) にする
<code>-t</code>	<code>--timeout</code>	ブートマネージャのタイムアウト (秒)
<code>-T</code>	<code>--delete-timeout</code>	タイムアウト変数を削除する

```
例 efibootmgr -v
例 efibootmgr -n 0003
例 efibootmgr -o 0003,0000,0002
例 efibootmgr -c -d /dev/sda -p 1 -L "Linux" -l
'\EFI\linux\grubx64.efi'
例 efibootmgr -b 0004 -B
```

オプション無しの `efibootmgr` も一覧を出しますが、各エントリのローダーのパスは出ません (パスまで見るなら `-v`)。

`-n` (`BootNext`) は次の 1 回だけ優先させ、起動すると変数は消えます。
`-o` (`BootOrder`) は次回以降すべてに効く順序です。
`-o` の引数はカンマ区切りの 1 つの文字列です。`-o 0003 0000 0002` と空白で区切ると意図した順序になりません。`-n` は単一のエントリ番号を取ります。

大文字は「削除」側です。
`-o` / `-O` は `BootOrder` の設定と、変数そのものの削除です。
`-b` / `-B`、`-n` / `-N`、`-t` / `-T` も同じ関係で、1 文字違いで意味が逆の誤答肢になります。

⚠ `efibootmgr` は Linux 起動後のコマンドで、`grub rescue>` からは使えません。BIOS レガシーブートには NVRAM のブートエントリ自体がありません。

1-2-9 `efibootmgr -v` の読み方と ESP の中身

`efibootmgr -v` は、NVRAM のブートエントリを詳しく出します：

```
BootCurrent: 0001
Timeout: 3 seconds
BootOrder: 0001,0000,0002
Boot0000* UEFI Shell      FvVol(...)/FvFile(...)
Boot0001* debian
HD(1,GPT,6a3b4c1e-...,0x800,0x100000)/File(\EFI\debian\grubx64.efi)
Boot0002 Network Boot    PciRoot(0x0)/Pci(0x1,0x0)/MAC(...)
```

- `BootCurrent` : 今回の起動に使われたエントリ
- `BootOrder` : ファームウェアが試す順。この例で次の通常起動に最初に試されるのは `Boot0001 (debian)`
- 名前の前の `*` : 有効なエントリのしるし。`Boot0002` のように `*` が無いものは無効
- `HD(1,GPT,<GUID>,<開始セクタ>,<サイズ>)` : ディスク上の何番目のパーティション(ここでは ESP)か
- `File(\EFI\debian\grubx64.efi)` : 読み込むローダーのパス。区切りは `\`

復旧のコマンドは、ディスク・パーティションの番号・ラベル・ローダーを指定します。NVMe の ESP が `/dev/nvme0n1p1` なら、次のとおりです：

```
efibootmgr -c -d /dev/nvme0n1 -p 1 -L "debian" -l
'\EFI\debian\grubx64.efi'
```

`-d` の既定は `/dev/sda`、`-p` の既定は `1` です。`-l` の既定はディストリビューション固有で、Rocky 9 は `\EFI\rocky\grub.efi` でした。`\` はシェルに解釈されないよう、**引用符で囲みます**。作ったら `efibootmgr -v` で確かめ、順序は `-o` で整えます。

ESP のファイル構成は、次の表のとおりです：

場所	中身
<code>/boot/efi/EFI/<ディストリ名>/</code>	そのディストリビューションの <code>shim…efi</code> 、 <code>grub…efi</code> 、小さな <code>grub.cfg</code>
<code>/boot/efi/EFI/BOOT/BOOTX64.EFI</code>	NVRAM にエントリが無くても読まれる 既定のパス (<code>x86_64.ARM64</code> は <code>BOOTAA64.EFI</code>)

- Rocky 9 の `x86_64`：`/boot/efi/EFI/rocky/` に `shimx64.efi`・`grubx64.efi`・`grub.cfg` が入る
- ARM64 版：`shimaa64.efi` と `grubaa64.efi` (`rpm -ql shim-aa64 grub2-efi-aa64` で確認)
- Debian 系：`/boot/efi/EFI/debian/`
- ESP の `grub.cfg`：小さな設定。本体の設定は `/boot/grub2/grub.cfg` (Debian 系は `/boot/grub/grub.cfg`)
- ESP の作り方：**FAT32** で作り、通常は `/boot/efi` にマウントする

1-3 GRUB 2 [202.2]

GRUB (GRand Unified Bootloader) は、最も広く使われるブートローダーです。BIOS でも UEFI でも使え、複数の OS を選んで起動できます。

1-3-1 GRUB Legacy と GRUB 2

GRUB には **GRUB Legacy (0.9x 系)** と、書き直された **GRUB 2 (1.9x 以降)** があります。試験は GRUB 2 が中心です (Legacy との差が対比で出る)。

	GRUB Legacy	GRUB 2
設定ファイル	<code>/boot/grub/menu.lst</code> (<code>grub.conf</code> へのリンクを持つ場合もある)	<code>/boot/grub2/grub.cfg</code> (Debian 系は <code>/boot/grub/grub.cfg</code>)
設定ファイルの扱い	管理者が直接編集する	生成物。直接編集しない
生成コマンド	無し(手で書く)	<code>grub2-mkconfig / grub-mkconfig</code> (Debian 系のラッパーは <code>update-grub</code>)
ディスクの番号	<code>hd0</code> から (0 起点)	<code>hd0</code> から (0 起点。ここは同じ)
パーティションの番号	0 起点 (1 番目のパーティションが <code>0</code>)	1 起点 (1 番目のパーティションが <code>1</code>)
ルートの指定	<code>root (hd0,0)</code>	<code>set root=(hd0,1)</code>
設定の書き方	キーワードの羅列	シェルスクリプト風(変数・条件分岐・関数が使える)

最も問われるのはパーティション番号の起点です。`/dev/sda1` を指すなら、**GRUB Legacy は `(hd0,0)` 、GRUB 2 は `(hd0,1)`** と書きます。ディスクの側

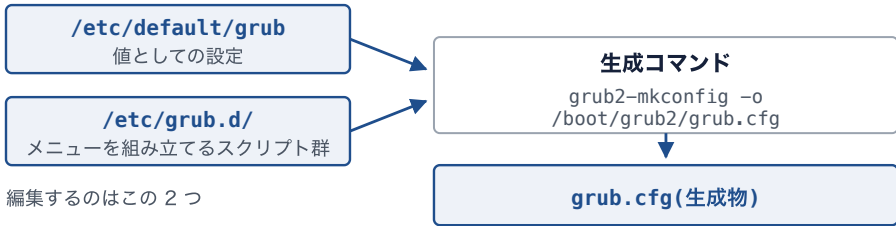
はどちらも 0 起点のまま変わりません。GRUB 2 では、形式を明示した `(hd0,msdos1)` (MBR) や `(hd0,gpt1)` (GPT) も使えます。

1-3-2 設定の三層：どこを直せば良いのか

GRUB 2 の設定は、「編集する場所」と「生成される場所」が分かれているのが最大の特徴です。

`/etc/default/grub` … 全体に効く変数（タイムアウト・既定エントリ・カーネルパラメータ）
+
`/etc/grub.d/` … メニュー項目を出力するスクリプト群（番号順に実行される）
↓ `grub2-mkconfig`（内部で `/etc/grub.d/` の各スクリプトを実行し、出力を連結）
`/boot/grub2/grub.cfg` … GRUB が実際に読む設定ファイル（生成物）

grub.cfg は「書くもの」ではなく「生成物」



grub2-install /dev/sda

ブートローダー本体をデバイスへ設置

grub2-mkconfig -o ...

設定ファイル grub.cfg を生成する

update-grub

Debian/Ubuntu 系のラッパー(RHEL 系には無い)

grub.cfg を手編集してはいけないのは、カーネル更新時の再生成で消えるから。

-o を付けずに打つと、生成結果は標準出力に表示されるだけでファイルは更新されない。

図 1-4 grub.cfg は生成物。編集するのは /etc/default/grub と /etc/grub.d/

● **grub.cfg は生成物で、編集対象ではありません。**カーネルやパッケージの更新で `grub2-mkconfig` が自動実行され、`grub.cfg` は丸ごと作り直されます。手で書いた内容は残りません。設定は `/etc/default/grub` か `/etc/grub.d/` の側で変えます。

理由は「書けないから」ではなく「上書きされるから」です。「バイナリ形式で開けない」「編集するとブートローダーが消える」「root でも書けない」は、どれも誤りです(テキストファイルで、権限もある)。

1-3-3 grub-mkconfig / grub2-mkconfig : 設定ファイルを生成する

⚠ **Red Hat 系は grub2- の接頭辞が付き、Debian 系は付きません**(中身は同じ GRUB 2)。

	Red Hat 系	Debian 系
設定生成	<code>grub2-mkconfig</code>	<code>grub-mkconfig</code> / ラッパーの <code>update-grub</code>
本体の導入	<code>grub2-install</code>	<code>grub-install</code>
設定ファイルの場所	<code>/boot/grub2/grub.cfg</code>	<code>/boot/grub/grub.cfg</code>
編集する変数ファイル	<code>/etc/default/grub</code>	<code>/etc/default/grub</code> (ここは共通)

`grub-mkconfig` / `grub2-mkconfig` : GRUB 2 の設定ファイルを生成する

書式 `grub2-mkconfig [-o <出力先>]` (Red Hat 系)

書式 `grub-mkconfig [-o <出力先>]` (Debian 系)

オプション	長形	意味
<code>-o</code>	<code>--output=FILE</code>	生成した設定内容を FILE へ書き出す(既定は標準出力)
<code>-h</code>	<code>--help</code>	使い方を表示して終了する
<code>-V</code>	<code>--version</code>	バージョンを表示して終了する

例 `grub2-mkconfig -o /boot/grub2/grub.cfg`

例 `grub-mkconfig -o /boot/grub/grub.cfg`

 `-o` を付けないと、生成内容は標準出力に出るだけで `grub.cfg` は更新されません。パスだけを渡す `grub2-mkconfig /boot/grub2/grub.cfg` も出力先に

なりません。

このコマンドは、設定の生成だけを行います。

- 本体の書き込みは `grub-install`
- `modules.dep` は `depmod`
- `initramfs` は `dracut` / `update-initramfs`
- 再起動も対話的な編集もしない

Debian 系 の `update-grub` は、内部で `grub-mkconfig -o /boot/grub/grub.cfg` を実行するだけのシェルスクリプトです(引数なしの 1 語で完結する)。

⚠ `update-grub` は Debian / Ubuntu 系 だけにあります。「2」付きの `grub2-mkconfig` は Red Hat 系 だけの名前です。Debian 系には、`update-grub` を呼ぶだけの互換用 `update-grub2` (`grub2-common` パッケージ) もあります。

1-3-4 `/etc/default/grub` の主な変数

`/etc/default/grub` は、シェルの変数代入の形で書くテキストファイルです(変数=値)。

```
GRUB_TIMEOUT=5
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="crashkernel=auto rd.lvm.lv=rhel/root rhgb quiet"
GRUB_DISABLE_RECOVERY="true"
GRUB_DISABLE_OS_PROBER=false
```

変数	何をするか	よくある既定値
GRUB_DEFAULT	既定で選択されるエントリ。 数字(0 起点)、エントリ名、 saved (前回選んだものを覚える)	0 または saved
GRUB_CMDLINE_LINUX	すべてのメニュー項目のカーネル行に追加されるカーネルパラメータ	ディストリビューション依存

● カーネルパラメータを恒久的に足すには、2 手が必要です。

1. `/etc/default/grub` の `GRUB_CMDLINE_LINUX` にパラメータを書き加える
2. `grub2-mkconfig -o /boot/grub2/grub.cfg` で `grub.cfg` を生成し直す

どちらか片方では効きません。対して、**メニューで `e` を押して書き足すのは、その回の起動だけの一時的な変更**です(1-3-7)。また、`/etc/modprobe.d/` はカーネルモジュールへのパラメータです。GRUB がカーネルに渡す起動時パラメータとは別系統です(`dracut -f` や `grub2-install` も無関係)。

1-3-5 /etc/grub.d/ と os-prober

/etc/grub.d/ には、番号の付いた実行可能なスクリプトが並んでいます。
grub2-mkconfig はこれを 番号順 に実行し、その標準出力を連結して grub.cfg を作ります。

スクリプト	何を出力するか
30_os-prober	os-prober を呼び出し、他パーティションにある OS のメニュー項目を出す
40_custom	管理者が自由に書き足すためのファイル(ここに書いた内容は再生成後も残る)

マルチブートで Windows や他のディストリビューションがメニューに現れるのは、os-prober の働きです。他のパーティションを走査して OS を検出し、30_os-prober を通じて grub.cfg に反映します (efibootmgr ・ update-initramfs ・ depmod は他 OS を探さない)。独自のメニュー項目は 40_custom に書けば、再生成でも消えません。

1-3-6 grub-install / grub2-install :ブートローダー本体を導入する

grub-install / grub2-install :GRUB 本体をデバイスへインストールする

書式	grub2-install [オプション] [インストール先デバイス]	(Red Hat 系)
書式	grub-install [オプション] [インストール先デバイス]	(Debian 系)

オプション	意味
<code>--target=TARGET</code>	導入先のプラットフォーム (<code>i386-pc</code> が BIOS 用の既定、 <code>x86_64-efi</code> が 64 ビット UEFI 用)
<code>--efi-directory=DIR</code>	ESP のルートとして使うディレクトリ (UEFI のときに指定する。通常 <code>/boot/efi</code>)

```
例  grub-install /dev/sda
例  grub2-install --target=x86_64-efi --efi-directory=/boot/efi --
    bootloader-id=redhat
```

● **BIOS／MBR 環境の導入先は、パーティションではなくディスク全体です** (`grub2-install /dev/sda` 。 `/dev/sda1` は誤り)。MBR はディスク全体の先頭にあるからです。UEFI 環境では `--target=x86_64-efi` と `--efi-directory` を付けて、ESP に `.efi` を置きます。

● `grub-install` と `grub-mkconfig` の役割は、まったく別です。

	<code>grub-install</code> (<code>grub2-install</code>)	<code>grub-mkconfig</code> (<code>grub2-mkconfig</code>)
何をする	ブートローダー本体 (<code>boot.img</code> ・ <code>core.img</code> などのコードとモジュール) を配置する	設定ファイル <code>grub.cfg</code> を生成する
配置先	BIOS/MBR なら MBR とその後ろ、UEFI なら ESP 上	<code>-o</code> で指定したパス (通常 <code>/boot/grub2/grub.cfg</code>)
<code>grub.cfg</code> を作るか	作らない	作る

「BIOS なら `grub-install`、UEFI なら `grub-mkconfig`」は誤りです。どちらの環境でも両方使います (**install は本体、mkconfig は設定**)。

1-3-7 GRUB 2 のメニューと一時的な編集 (e キー)

キー	何が起きるか
↑ ↓	エントリの選択
Enter	選択したエントリで起動
e	選択したエントリの内容をその場で編集する (編集画面へ入る)
c	GRUB のコマンドライン (grub>)へ入る
Esc	編集画面からメニューへ戻る
Ctrl + x (または F10)	編集した内容でそのまま起動する

e を押すと、選択中の項目 (`linux` 行と `initrd` 行を含む) を書き換えられます。カーネルパラメータを一時的に試すなら、`linux` 行の末尾に書き足して **Ctrl + x** で起動します。この編集はメモリ上だけで、`grub.cfg` には保存されません。

起動の最初から復旧用のターゲットへ入るのも、この編集です。`linux` 行の末尾に次を足します。

<code>systemd.unit=rescue.target</code>	… <code>rescue.target</code> で起動する
<code>systemd.unit=emergency.target</code>	… <code>emergency.target</code> で起動する
<code>init=/bin/bash</code>	… <code>init</code> の代わりに直接シェルを起動する
<code>single</code>	… 互換のシングルユーザー指定 (<code>rescue.target</code> 相当へ導かれる)

`systemctl` を使わなくても、この方法で起動時から `emergency.target` に入れます(「メニューからは既定のターゲットでしか起動できない」は誤り)。カーネル更新後に起動しなくなったら、メニューに残る更新前のカーネルを選びます。

1-3-8 `grub>` と `grub rescue>` :GRUB のプロンプトから復旧する

GRUB がメニューを出せなかったとき、二種類のプロンプトのどちらかで止まります。

プロンプト	いつ出るか	使えるもの
<code>grub></code>	GRUB 本体 (<code>normal</code> モジュールを含む通常モード) は読み込めたが、 <code>grub.cfg</code> が無い・エントリが選べない	GRUB の全コマンドがほぼ使える
<code>grub rescue></code>	<code>grub.cfg</code> の欠落・破損や、モジュールを置いたパーティションが見つからず、通常モードに入れなかった	ごく限られた内蔵コマンドだけ

`grub rescue>` は最小限のシェルです。使えるのは `ls` ・ `set` ・ `unset` ・ `insmod` ・ `normal` ・ `boot` など数えるほどしかありません。

grub rescue> の内蔵コマンド:最小シェルからの手動復旧

書式	ls	デバイスとパーティションを一覧する
書式	ls (hdX,Y)/	そのパーティションの中身を見る
書式	set	現在の変数を表示する
書式	set root=(hdX,Y)	GRUB が基準にするパーティションを設定する
書式	set prefix=(hdX,Y)/boot/grub2	モジュールの置き場を設定する
書式	insmod normal	通常モードのモジュールを読み込む
書式	normal	通常モードへ復帰する（メニューが出る）
書式	linux /vmlinuz-<版> root=/dev/sdaN	カーネルを読み込む
書式	initrd /initramfs-<版>.img	initramfs を読み込む
書式	boot	読み込んだカーネルで起動する

例 set root=(hd0,1)

例 set prefix=(hd0,1)/boot/grub2

ls でパーティションを確かめ、**set root=(hdX,Y)** で当たりを付けます。続けて **insmod normal** と **normal** で通常モードへ戻るのが型です。起動できたら、Linux 上で **grub2-mkconfig** を打って **grub.cfg** を作り直します（必要なら **grub2-install** で本体も入れ直す）。

 **grub rescue>** は Linux ではありません。 **grub2-mkconfig** ・ **efibootmgr** ・ **mount** ・ **chroot** ・ **systemctl** はどれもこのプロンプトにはありません。

1-3-9 レスキューメディアからの復旧: `mount` → `chroot` → `grub-install`

`grub rescue>` からも起動できないときは、インストールメディア(レスキューISO)で起動し、ディスク上のシステムに `chroot` してから修復します。

```
例 mount /dev/sda2 /mnt
例 mount /dev/sda1 /mnt/boot
例 mount --bind /dev /mnt/dev
例 mount --bind /proc /mnt/proc
例 mount --bind /sys /mnt/sys
例 chroot /mnt
例 grub2-install /dev/sda
例 grub2-mkconfig -o /boot/grub2/grub.cfg
例 exit
例 umount -R /mnt
```

手順	何のために
① <code>mount <ルート of デバイス> /mnt</code>	ディスク上のルートファイルシステムを、レスキュー環境の <code>/mnt</code> に見せる
② <code>/boot</code> (UEFI なら <code>/boot/efi</code> も) を追加でマウント	カーネル・initramfs・ESP が <code>chroot</code> の中から見えるようにする
③ <code>mount --bind</code> で <code>/dev</code> ・ <code>/proc</code> ・ <code>/sys</code> を持ち込む	<code>grub-install</code> はデバイスを実際に読み書きするので、これが無いと失敗する
④ <code>chroot /mnt</code>	ここから先のコマンドが、ディスク上のシステムを自分のルートとして扱う

手順	何のために
⑤ <code>grub2-install <ディスク></code>	ブートローダー本体を入れ直す(ディスク全体を指定。パーティションではない)
⑥ <code>grub2-mkconfig -o <grub.cfg のパス></code>	メニューの設定ファイルを作り直す
⑦ <code>exit</code> → <code>umount -R /mnt</code>	<code>chroot</code> を抜け、まとめてアンマウントしてから再起動する

● `--bind` の 3 つが抜けると、`grub-install` が「デバイスが見つからない」で止まります。UEFI では ESP (`/boot/efi`) もマウントし、必要なら `chroot` の中で `efibootmgr` でエントリを作り直します。

⚠ `fsck` は、アンマウントの状態で `mount` の前に走らせます (マウント中にかけると壊しかねない)。

1-3-10 NVMe から起動する

NVMe SSD は、SATA の `/dev/sd*` とは別の系統の名前です。

見え方	何を指すか
<code>/dev/nvme0</code>	1 台目の NVMe コントローラ
<code>/dev/nvme0n1</code>	そのコントローラの 1 番目のネームスペース (SATA でいうディスク 1 台に相当)
<code>/dev/nvme0n1p1</code>	そのネームスペースの第 1 パーティション

● パーティション番号の前に `p` が入ります (`/dev/nvme0n1p1` 。 `/dev/nvme0n11` は誤り) 。 `n` の後ろはネームスペース番号です。

起動の流れは同じです。UEFI では、ESP を `/dev/nvme0n1p1` に置いて `/boot/efi` にマウントし、`grub2-install` を実行します。古い BIOS 専用機は、NVMe から直接起動できないことがあります。GRUB は NVMe 上の `/boot` も `(hd0,gpt2)` のような通常の表記で扱うので、`grub.cfg` に NVMe 固有の指定はありません。

例 `ls /dev/nvme*`

例 `mount /dev/nvme0n1p1 /boot/efi`

例 `grub2-install --target=x86_64-efi --efi-directory=/boot/efi`

⚠ NVMe の名前も順番が変わり得るので、`/etc/fstab` では UUID で書きます(3-2)。`nvme` コマンドは 4-8 で扱います。

1-3-11 `grub.cfg` のメニュー項目を読む

生成された `grub.cfg` の 1 項目は、次のような形です (Debian 系の例。`search` などの行は省いています)。

```
menuentry 'Debian GNU/Linux' --class debian --class gnu-linux --class
gnu --class os {
    insmod lvm
    insmod ext2
    echo      'Loading Linux 6.1.0-53-arm64 ...'
    linux     /boot/vmlinuz-6.1.0-53-arm64 root=/dev/mapper/vgr-
root ro quiet
    echo      'Loading initial ramdisk ...'
    initrd    /boot/initrd.img-6.1.0-53-arm64
}
```

行	意味
<code>insmod</code>	GRUB 自身のモジュール を読む(ここでは LVM と ext2 を読む部分)。カーネルのモジュールとは別物
<code>linux</code>	カーネルのパス+カーネルに渡す引数。 <code>root=</code> はカーネルから見たルート。 <code>ro</code> は最初に読み取り専用でマウントする指定
<code>initrd</code>	initramfs のパス。 カーネルの起動直後に、メモリ上へ展開される

- `ro` で始める理由: `fsck` を安全に走らせるため。その後、`systemd` が読み書き可能にする
- メニューの表示時間: `GRUB_TIMEOUT` と `GRUB_TIMEOUT_STYLE` で決まる

RHEL 系は名前が異なります。

- `initramfs`: `initramfs-<版>.img`
- Rocky 9 のような **BLS (Boot Loader Specification)** の系統:カーネルごとの項目が、`/boot/loader/entries/` の**別々のファイル**に分かれる
- `grub.cfg`: `blscfg` の行が書かれ、そこで読み込まれる

1-3-12 カーネル行の引数:通常の項目と recovery の項目

`/etc/default/grub` の 2 つの変数は、recovery mode の項目への入り方が違います:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
GRUB_CMDLINE_LINUX="console=ttyS0,115200"
```

項目	カーネル行に付く引数
通常のエントリ	<code>GRUB_CMDLINE_LINUX</code> と <code>GRUB_CMDLINE_LINUX_DEFAULT</code> の両方
recovery mode のエントリ	<code>GRUB_CMDLINE_LINUX</code> だけ (<code>single</code> が加わる)

`10_linux` のスクリプトが、このとおりに組み立てています (Debian 12 で確認)。

- 画面の出力を抑える `quiet splash` は、`DEFAULT` 側に置く。障害時の `recovery mode` ではメッセージが見える
- シリアルコンソールのように常に必要な引数は、`GRUB_CMDLINE_LINUX` 側に置く
- 設定を変えたら、`grub-mkconfig` で作り直す (Debian 系は `update-grub`、RHEL 系は `grub2-mkconfig`)

1-3-13 /etc/grub.d/ の動かし方と、既定エントリの切り替え

`grub-mkconfig` は、`/etc/grub.d/` の実行権限があるスクリプトだけを、ファイル名の番号の小さい順に実行します (Debian 12 の `grub-mkconfig` で `test -x` の判定を確認)。

```
00_header 05_debian_theme 10_linux 20_linux_xen 30_os-prober
30_uefi-firmware 40_custom 41_custom
```

- `40_custom` の `menuentry` は、`10_linux` が出す項目の後ろに入る
- 使わないスクリプトは `chmod -x` で外す。外すと、出力は `grub.cfg` から消える (権限を戻して再実行すれば復活)

- `grub.cfg` は生成物なので、手では編集しない。独自の項目は `40_custom`、既定値の変更は `/etc/default/grub` に書く

次回の起動だけ別の項目にするには、既定を `saved` にします：

```
GRUB_DEFAULT=saved          ← /etc/default/grub に書き、grub-mkconfig
                             を実行する
grub-set-default 1          ← 既定を 1 番目 (0 起点) に保存する
grub-reboot 2               ← 次の 1 回だけ 2 番目で起動する
grub-editenv list
saved_entry=1
next_entry=2
```

値	意味
<code>saved_entry</code>	次回以降の既定。 <code>grub-set-default</code> が書く
<code>next_entry</code>	1 回だけ 使う指定。 <code>grub-reboot</code> が書き、使った後は消える。失敗しても次は元の既定に戻る

値は環境ブロックの `grubenv` に入ります。

- 場所: Debian 系は `/boot/grub/grubenv`、RHEL 系は `/boot/grub2/grubenv`
- 読み書き: `grub-editenv`
- RHEL 系のコマンド名: `grub2-set-default` • `grub2-reboot` • `grub2-editenv`

`grub-editenv` で `saved_entry=1` と `next_entry=2` を設定し、`list` で読み出せることは、Debian 12 で確認しました。

1-3-14 RHEL 系の UEFI で ESP を戻すとき

RHEL 系の UEFI では、`grub2-install` を使いません。`grub2-efi-x64` と `shim-x64` のパッケージを**再インストール**して、ESP のファイルを戻すのが標準です。BIOS の機械では `grub2-install` を使います。

- UEFI で `grub-install` を実行すると、ESP にファイルを置き、NVRAM の起動エントリも登録する
- `--bootloader-id=<名前>` :EFI のみの指定。ESP に作るディレクトリとエントリの名前になる

1-3-15 `grub>` から手動で起動する手順

`grub>` で止まったら、まず**どのパーティションに何があるか**を調べます：

```
grub> ls
(hd0) (hd0,gpt2) (hd0,gpt1)
grub> ls (hd0,gpt1)/
EFI/
grub> ls (hd0,gpt2)/boot/
grub/  vmlinuz-6.1.0-47-amd64  initrd.img-6.1.0-47-amd64
```

`EFI/` だけのパーティションは ESP です。`vmlinuz-...` と `initrd.img-...` があるのが、`/boot` を含むルートです。ルートが `/dev/sda2` なら、次の順に打ちます：

```
set root=(hd0,gpt2)

linux /boot/vmlinuz-6.1.0-47-amd64 root=/dev/sda2

initrd /boot/initrd.img-6.1.0-47-amd64

boot
```

- **set root=(hd0,gpt2)** : GRUB がファイルを読むパーティションの指定
- **root=/dev/sda2** (**linux** 行) : カーネルに渡す引数

別物なので、両方が要ります。GRUB 2 のパーティション番号は 1 起点です。

(hd0,msdos1) は MBR、(hd0,gpt2) は GPT の表記です。

起動できたら、OS 側で **grub-mkconfig** と **grub-install** を実行して、設定を直します。

通常モードのメニューへ戻すには、**grub rescue>** で次の順に打ちます (1-3-8)。

- **set prefix=(hd0,gpt2)/boot/grub**
- **insmod normal**
- **normal**

1-4 initramfs: ルートをマウントするための一時的な環境 [202.2]

1-4-1 なぜ必要なのか

カーネルは起動直後、**まだルートファイルシステムをマウントできていません**。ルートが LVM・LUKS・ソフトウェア RAID の上にあると、マウントに要るモジュールと道具がルートの中にあって取り出せません (組み込まれていないス

トレージコントローラも同じ)。**initrd**／**initramfs** は、この鶏と卵を解く一時的なルートファイルシステムです。

- ブートローダーがカーネルと一緒にメモリへ読み込む
- 中身はカーネル自身が展開して、一時的なルートファイルシステムとして使う
- 必要なモジュール(ストレージドライバ・LVM・RAID・LUKS)と最小限の道具(BusyBox など)が入っている
- 本来のルートをマウントできたら、**switch_root (initrd 形式では pivot_root)**で制御を移し、役目を終える

展開するのはファームウェアではなくカーネルです(ブートローダーはメモリへ載せるまで)。

initrd はブロックデバイスとして扱う古い方式(初期 RAM ディスク)です。**initramfs** は **cpio** アーカイブを **tmpfs** へ展開する新しい方式です。今のディストリビューションは実質 **initramfs** だけですが、**initrd.img-<版>** のような旧名のファイルは残ります。試験でも両方の呼び方が出ます。NVRAM のブートエントリ、MBR、**grub.cfg** の再生成、インターフェース名、スワップは、どれも **initramfs** の役割ではありません。

1-4-2 **dracut** :Red Hat 系の **initramfs** 生成

dracut :**initramfs** イメージを生成する

書式 **dracut** [オプション] [<出力イメージ> [<カーネルバージョン>]]

オプション	長形	意味
<code>-f</code>	<code>--force</code>	既存の <code>initramfs</code> ファイルを上書きする
<code>-H</code>	<code>--hostonly</code>	このホストに必要なものだけを入れる(既定)
<code>-N</code>	<code>--no-hostonly</code>	ホスト限定モードを無効にし、汎用的に作る
<code>-o</code>	<code>--omit <モジュール…></code>	指定した <code>dracut</code> モジュールを除外する

例 `dracut -f`

例 `dracut --force --regenerate-all`

例 `dracut -f /boot/initramfs-$(uname -r).img $(uname -r)`

例 `dracut --add-drivers "mpt3sas" -f`

`-f` は既存の `initramfs` の上書きです(ファイルは既にあるのが普通で、付けないと「既に存在する」で何もしない)。`-H (hostonly)` が既定で、このホストで使うドライバだけを入れるので小さく収まります。ただし、別のハードウェアへ持って行くと起動しないことがあります。汎用のイメージには `-N ( --no-hostonly )` を使います(ホスト限定をやめる)。

1-4-3 `mkinitramfs` / `update-initramfs` : Debian 系の `initramfs` 生成

Debian 系は `initramfs-tools` パッケージ(設定は `/etc/initramfs-tools/`)です。低レベルの `mkinitramfs` と、それを包む `update-initramfs` の二段構えです。

mkinitramfs :initramfs イメージを作る(低レベル)

書式 `mkinitramfs [オプション]... -o <出力ファイル> [<カーネルバージョン>]`

オプション	意味
<code>-o <出力ファイル></code>	生成したイメージの書き出し先(必須)

例 `mkinitramfs -o /boot/initrd.img-$(uname -r) $(uname -r)`

例 `mkinitramfs -o /tmp/initrd.img 6.1.0-18-amd64`

update-initramfs :initramfs をまとめて作成・更新・削除する

書式 `update-initramfs {-c|-d|-u} [-k <カーネルバージョン>] [-v] [-b <ディレクトリ>]`

オプション	意味
<code>-c</code>	新しい initramfs を 作成 する
<code>-u</code>	既存の initramfs を 更新 する
<code>-d</code>	既存の initramfs を 削除 する
<code>-k <版></code>	対象のカーネルバージョン。all ですべての導入済みカーネルを対象にする

例

update-initramfs -u

例

update-initramfs -u -k all

例

update-initramfs -c -k 6.1.0-18-amd64

例

update-initramfs -d -k 6.1.0-13-amd64

`-c`・`-u`・`-d` のどれか 1 つは必須です。日常は `-u` で、`-k all` を添えると導入済みのカーネル全部を作り直します。**⚠** `update-initramfs` は自分の `initramfs` だけを更新します (他の OS の検出や GRUB の設定には触れない)。

コマンド	系統	何を作るか
<code>dracut</code>	Red Hat 系	<code>initramfs</code>
<code>mkinitrd</code>	古い Red Hat 系 (<code>dracut</code> の前身)	<code>initrd</code>
<code>mkinitramfs</code>	Debian 系	<code>initramfs</code> (1 個ずつ)
<code>update-initramfs</code>	Debian 系	<code>initramfs</code> (まとめて。内部で <code>mkinitramfs</code> を呼ぶ)
<code>grub2-mkconfig</code> / <code>update-grub</code>	Red Hat 系 / Debian 系	GRUB の設定ファイル (<code>initramfs</code> ではない)

1-4-4 initramfs 内のシェル

ルートのマウントに失敗すると、`initramfs` は同梱の BusyBox シェル (`(initramfs)` のプロンプト) で止まります。カーネルは動いているが、ルートへ切り替わる前の段階です。モジュールを手で読み込み、`ls /dev/` や `blkid` でデバイス名や UUID を確かめます (UEFI シェルや `grub rescue>` とは階層が違う)。

1-4-5 initramfs の中身を調べる・足す

起動が「ルートのデバイスが見つからない」で止まるときは、必要なドライバが initramfs に入っているかを調べます：

目的	Debian 系	RHEL 系 (dracut)
中身を一覧する	<code>lsinitramfs <イメージ></code>	<code>lsinitrd <イメージ></code>
展開する	<code>unmkinitramfs <イメージ> <出力先></code>	<code>lsinitrd -f <ファイル> <イメージ></code> (特定のファイルだけ表示)
作り直す	<code>update-initramfs -u</code>	<code>dracut -f</code>
必須でないモジュールを足す	<code>/etc/initramfs-tools/modules</code> に名前を 1 行書き、 <code>update-initramfs -u</code>	<code>/etc/dracut.conf.d/</code> に <code>add_drivers+= " xfs "</code> と書き、 <code>dracut -f</code>

```
# lsinitramfs /boot/initrd.img-6.1.0-47-cloud-arm64 | grep -E
'virtio_blk|scripts/local$'
scripts/local
usr/lib/modules/6.1.0-47-cloud-arm64/kernel/drivers/block/virtio_blk.ko
```

一覧に `virtio_blk.ko` が出れば、ドライバは入っています。出なければ、上の表の方法で足して作り直します。

`dracut` の主なオプションは、次のとおりです (Rocky 9 の `dracut --help` で確認)。

- `--kver` : カーネルの版を指定する

- `-f` (`--force`) : 既存のファイルを上書きする
- `--add-drivers` : ドライバを追加する
- `--regenerate-all` : 既定の場所にあるすべての版を作り直す

既定の設定は `/usr/lib/dracut/dracut.conf.d/01-dist.conf` にあります。管理者の設定は `/etc/dracut.conf.d/` に置きます。旧来の `mkinitrd` は、`dracut` に置き換えられました。

1-4-6 止まった場所を、メッセージから見分ける

起動の流れは、ファームウェア → ブートローダー → カーネル → `initramfs` → `systemd` です。**どの段で止まったかは、メッセージで分かります**(`boot(7)`)。

止まった段	見えるメッセージの例	疑うこと
ファームウェア	起動デバイスが見つからない	起動順、ディスク、ESP
ブートローダー	<code>error: no such partition</code> 、 <code>grub rescue></code>	<code>grub.cfg</code> の欠落・破損
カーネル	<code>Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)</code>	ルートを認識できない。 <code>initramfs</code> のドライバ不足、 <code>root=</code> の誤り
<code>initramfs</code>	<code>dracut</code> の <code>Dropping to debug shell</code> 、 <code>BusyBox</code> の (<code>initramfs</code>)	ルートの待ち合わせに失敗
<code>systemd</code>	<code>Welcome to emergency mode</code>	<code>/etc/fstab</code> の誤り、ファイルシステムの破損

`Kernel panic ... unknown-block(0,0)` は、「カーネルは起動したが、ルートのブロックデバイスを見つけられない」という意味です。GRUB で `root=` の指定を確かめます。ライブ媒体から `initramfs` を作り直す手もあります。

1-4-7 保守モードで起動する引数と、読み書きの戻し方

カーネル行に付ける引数で、保守用の状態に入れます (`systemd(1)` の説明)。

引数	入る先
<code>emergency</code> ・ <code>rd.emergency</code> ・ <code>-b</code>	<code>emergency.target</code>
<code>rescue</code> ・ <code>rd.rescue</code> ・ <code>single</code> ・ <code>s</code> ・ <code>S</code> ・ <code>1</code>	<code>rescue.target</code>
<code>init=/bin/bash</code>	<code>systemd</code> の代わりに <code>bash</code> が PID 1 になる。サービスの起動や <code>/</code> 以外のマウントは行われない
<code>systemd.unit=emergency.target</code>	起動先のターゲットを直接指定する

`emergency` や `init=/bin/bash` では、`/` が**読み取り専用**のことが多くあります。`/etc/fstab` を直しても保存できません。

```
# touch /etc/test
touch: cannot touch '/etc/test': Read-only file system
```

`findmnt -no OPTIONS /` で `ro` を確かめ、次のように読み書きへ戻します：

```
mount -o remount,rw /
```

直した後の戻し方は、次のとおりです。

- `systemctl daemon-reload` のあとに `reboot` する
- または `systemctl default` で、通常の起動へ進む
- `init=/bin/bash` から出るときは、`exec /sbin/init` で続けるか、`sync` して強制的に再起動する
- `fsck` が必要ときは、読み取り専用のままか、未マウントの状態で実行する

`chroot` で修復する手順は、次のとおりです。

1. ルートを `/mnt` にマウントする
2. `/boot` が別のパーティションなら `/mnt/boot` に、UEFI なら ESP を `/mnt/boot/efi` にマウントする
3. `/dev` ・ `/proc` ・ `/sys` を `mount --bind` で持ち込む
4. `chroot /mnt` に入る

bind マウントが無いと、`/proc` が空で `mount` が動きません。終わったら `exit` して、マウントを逆順に外してから再起動します。

1-5 init:systemd と、その前の世代 [202.1]

ルートファイルシステムへ切り替わると、カーネルは**最初のユーザープロセス (PID 1)**を起動します。これが `init (PID 1)` です。ここから先が「サービスを立ち上げてシステムを使える状態にする」段階です。

1-5-1 三つの世代

方式	設定の中心	状態の呼び方	起動の仕方
SysVinit	<code>/etc/inittab</code> 、 <code>/etc/init.d/</code> のスクリプト、 <code>/etc/rc<N>.d</code> / のリンク	ランレベル (0~6)	スクリプトを番号順に1つずつ実行する (直列)
Upstart	<code>/etc/init/*.conf</code>	ジョブとイベント	イベント駆動 (デバイス検出などをきっかけに起動)
systemd	<code>/etc/systemd/system/</code> などのユニットファイル	ターゲット	依存関係を解決して並列に起動する

現在の主要ディストリビューションは、systemd を採っています (RHEL・Debian・Ubuntu など)。Upstart は Ubuntu と RHEL 6 で使われた中間の世代です。名前と考え方 (イベント駆動) が分かれば足ります。SysVinit のランレベルは、systemd に互換の対応付けとして残っています (1-5-5)。

1-5-2 ユニット:systemd が管理する単位

systemd は、管理する対象をユニットという単位で扱います。種類は、拡張子を見れば分かるようにできています (次の表)。

拡張子	何を表すか
<code>.service</code>	サービス (デーモン)。もっともよく使う
<code>.target</code>	複数のユニットをまとめた「到達したい状態」。ランレベルに相当する

拡張子	何を表すか
<code>.mount</code>	マウントポイント (<code>/etc/fstab</code> から自動生成もされる)
<code>.automount</code>	自動マウントポイント
<code>.socket</code>	ソケット。接続が来てから対応するサービスを起動できる
<code>.timer</code>	時刻やタイマーによる起動 (cron に相当)
<code>.swap</code>	スワップ領域

1-5-3 ユニットファイルの探索順とドロップイン

同じ名前のユニットファイルが複数の場所にあり得ます。**systemd は優先度の最も高い 1 つだけを採用し、基本のユニットファイルどうしはマージしません。**

優先度	ディレクトリ	誰が置くか
高	<code>/etc/systemd/system/</code>	管理者 によるローカル設定
中	<code>/run/systemd/system/</code>	実行時に生成されたもの (再起動で消える)
低	<code>/usr/lib/systemd/system/</code> (<code>/lib/systemd/system/</code>)	パッケージ (ベンダー) が提供する既定のユニットファイル

● **`/etc` が最優先、`/usr/lib` が最下位**です (管理者の設定がパッケージの既定を上書きする)。逆に書いた選択肢 (`/run` が `/etc` より上、`/usr/lib` が最優先) が定番です。

ドロップインは、ユニットの一部だけを上書き・追加するファイルです。`/etc/systemd/system/<ユニット名>.d/` に `.conf` を置くと、元の内容にその指定が重なります(丸ごとは置き換わらない)。一行だけ変えたいときの標準のやり方で、`systemctl edit <ユニット名>` がこの形のファイルを用意します。

1-5-4 ユニットファイルの中身と依存関係

サービスユニットの実物を載せます。

```
[Unit]
Description=The Apache HTTP Server
After=network.target remote-fs.target nss-lookup.target
Wants=httpd-init.service

[Service]
Type=notify
ExecStart=/usr/sbin/httpd $OPTIONS -DFOREGROUND
ExecReload=/usr/sbin/httpd $OPTIONS -k graceful
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

セクション	何を書くか
[Unit]	説明と 依存関係 (<code>After=</code> ・ <code>Before=</code> ・ <code>Wants=</code> ・ <code>Requires=</code> ・ <code>Conflicts=</code> など)。ユニット種別によらず共通

セクション	何を書くか
[Service]	そのサービス固有の内容 (Type= ・ ExecStart= ・ ExecStop= ・ Restart= ・ User= など) 。 .service の場合のセクション名がこれ
[Install]	systemctl enable したときに何が起こるか (WantedBy= ・ RequiredBy= ・ Alias=)

依存関係の指定子は、「強さ」と「順序」が別の軸になっているのがポイントです。

指定子	何を意味するか
Requires=	強い依存 。依存先の起動に失敗すると、このユニット自身の起動も失敗し得る
Wants=	弱い依存 。依存先の起動に失敗しても、このユニットは起動を続ける
BindsTo=	Requires= に加えて、 依存先が停止・失敗したら自分も停止する (継続的に連動する)
After=	順序 。指定したユニットの後に起動する
Before=	順序 。指定したユニットの前に起動する

 **Requires= も Wants= も、単独では起動順序を保証しません。** 順序は After= ／ Before= で別に分けます。「Wants= の列挙順に起動する」「Wants= があれば After= は不要」は、どちらも誤りです。なお、どちらも空白区切りで複数のユニットを並べられ、複数行にも分けられます。

ネットワークの設定後に動かしたいサービスは、`Wants=network-online.target` と `After=network-online.target` を組で書きます。`network.target` は、アドレスの設定まで済んだことを保証しません。

1-5-5 ターゲットとランレベル

ターゲットは、複数のユニットをまとめた「到達したい状態」で、SysVinit のランレベルに相当します。systemd は後方互換のために、**エイリアス** (`runlevel<N>.target` というシンボリックリンク)を用意しています。

ランレベル	ターゲット	状態
0	<code>poweroff.target</code>	シャットダウン(電源断)
1	<code>rescue.target</code>	シングルユーザー (レスキュー)
2・3・4	<code>multi-user.target</code>	テキストログインのマルチユーザー
5	<code>graphical.target</code>	グラフィカルログイン
6	<code>reboot.target</code>	再起動
—	<code>emergency.target</code>	緊急モード(ランレベルに対応するものが無い)

● `multi-user.target` がランレベル 3、`graphical.target` がランレベル 5 です。この 2 つを逆にした選択肢が定番です。また `rescue.target` はランレベル 1 (シングルユーザー) に相当するのであって、0 や 6 ではありません。

`runlevel1.target` のようなエイリアスも、`runlevel` コマンドも、systemd v257 以前 (RHEL 9・10 など) では互換のために残っています (ランレベルを完全に廃止した、は誤り)。なお v258 で既定から外れました。v260 以降は

`runlevel[0-6].target` だけをビルド時の指定で戻せます (`runlevel` ・ `telinit` コマンドは戻りません)。

ターゲットどうしにも依存関係があり、下の層から積み上がっていきます。

`emergency.target` … 何も積まない最小の状態

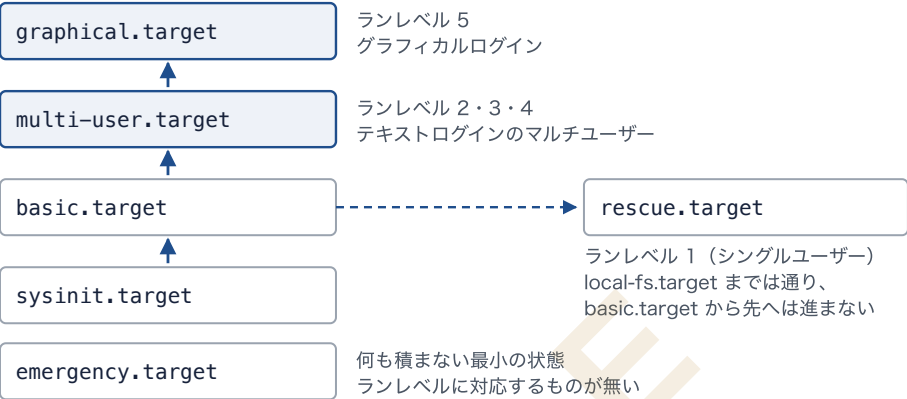
`sysinit.target` → `basic.target` → `multi-user.target` →
`graphical.target`

↓
└→ `rescue.target` (`local-fs.target` までは通り、`basic.target` から先へは進まない)

矢印は「積み上がる向き」です。`rescue.target` は、`multi-user.target` の手前で止まる状態です。そこから `multi-user.target` へ、自動的に進みません。進めたいときは `systemctl isolate multi-user.target` を打ちます。

systemd のターゲット —— 下から積み上がる

矢印は「積み上がる向き」。rescue.target は multi-user.target の手前で止まる状態。



multi-user.target がランレベル 3、graphical.target がランレベル 5。逆にした選択肢が定番。
分かれ目はマウント —— rescue.target はローカルファイルシステムをマウントし、
emergency.target はマウント処理をほぼ行わない。

図 1-5 systemd のターゲット：下から積み上がり、rescue.target は手前で止まる

1-5-6 rescue.target と emergency.target

トラブルシューティングで使う 2 つのターゲットは、**ファイルシステムをマウントするかどうか**で分かります。

	rescue.target	emergency.target
ランレベル対応	1（シングルユーザー）	対応なし
ファイルシステム	local-fs.target 等を経てローカルファイルシステムをマウントする	マウント処理をほぼ行わない（起動時に ro を渡していればルートは読み取り専用のまま）
通常のマルチユーザー向けサービス	起動しない	起動しない

	rescue.target	emergency.target
ネットワーク(sshd・NetworkManager)	起動しない	起動しない
位置づけ	シングルユーザーのメンテナンス環境	それよりさらに限定的な最小環境

● **分かれ目はマウントです。** `/etc` や `/var` が壊れて `rescue.target` にすら入れないときに `emergency.target` を使います。2 つは同じユニットの別名ではなく、別々のターゲットです。

1-5-7 `systemctl` : `systemd` を操作する

`systemctl` : ユニットの起動・停止・有効化・状態確認

書式 `systemctl` [オプション] <サブコマンド> [<ユニット名>...]

状態を変えるサブコマンド

サブコマンド	何をするか
<code>start</code>	いますぐユニットを起動する。次回起動時の設定は変わらない
<code>stop</code>	いますぐユニットを停止する。将来の起動は禁止しない
<code>restart</code>	停止してから起動し直す
<code>reload</code>	設定だけを読み直させる(そのサービスが対応している場合)

サブコマンド	何をするか
<code>enable</code>	次回起動時からの自動起動を有効にする (<code>[Install]</code> の <code>WantedBy=</code> 等に従い <code>.wants</code> ディレクトリへシンボリックリンク を作る)。いま起動はしない
<code>disable</code>	自動起動のシンボリックリンクを削除する。 手動の <code>start</code> は引き続きできる
<code>enable --now</code> / <code>disable --now</code>	有効化(無効化)と、起動(停止)を同時に 行う
<code>mask</code>	ユニット名から <code>/dev/null</code> へのシンボ リックリンクを作り、手動起動も依存関係経 由の起動も含めて一切を禁止する
<code>unmask</code>	<code>mask</code> が作った <code>/dev/null</code> へのリンクを 削除し、通常の状態へ戻す
<code>isolate</code>	指定したターゲットとその依存ユニットだ けを起動し、含まれない他のユニットを停 止する(いますぐの状態切り替え)
<code>set-default</code>	次回起動時の既定ターゲット (<code>default.target</code> のリンク先)を恒久的 に変更する
<code>daemon-reload</code>	ユニットファイルの変更内容を <code>systemd</code> に 再読み込みさせる
<code>reset-failed</code>	<code>failed</code> 状態のカウンタをリセットする
<code>rescue</code> / <code>emergency</code>	<code>isolate rescue.target</code> / <code>isolate</code> <code>emergency.target</code> の短縮形

調べるサブコマンド

サブコマンド	何を出すか
status	現在の実行状態 (Loaded: 行と Active: 行) と直近のログ
is-active	いま稼働中かどうか (active / inactive)。終了コードでも判定できる
is-enabled	次回起動時に自動起動する設定かどうか (enabled / disabled / static など)
cat	本体のユニットファイルとドロップインの内容をまとめて表示する
show	ユニットのプロパティ (内部的な設定値) を キー=値 の形で一覧する
list-dependencies	依存関係をツリー表示する

主なオプション

オプション	意味
--type=<種別>	ユニットの種別で絞る (service ・ target ・ mount ・ timer など)
--state=<状態>	状態で絞る (active ・ inactive ・ failed ・ enabled など)
--now	enable ／ disable に添えて、起動／停止も同時に行う

```
例 systemctl unmask telnetd
例 systemctl list-units --type=service --state=active
例 systemctl isolate rescue.target
例 systemctl set-default graphical.target
例 systemctl cat sshd.service
例 systemctl enable --now httpd.service
```

ユニット名の `.service` は省略できます (`systemctl mask telnetd` と `systemctl mask telnetd.service` は同じ)。`.target` や `.timer` は省けません。オプションの値は、`=` でも空白でも区切れます (`--type=service` と `--type service` は同じ)。

1-5-8 有効化・起動・禁止:4つの段差

`enable` と `start` は役割が逆になった選択肢が定番です。リンクを作って次回に備えるのが `enable`、いますぐプロセスを起こすのが `start` です。両方なら `enable --now` を使います (有効化と起動を同時に行う)。

`disable` と `mask` は「強さ」が違います。

	disable	mask
何をするか	自動起動用のシンボリックリンクを削除する	ユニット名から <code>/dev/null</code> へのシンボリックリンクを作る
手動の <code>systemctl start</code>	できる	できない (Unit is masked として拒否される)
他ユニットからの依存関係経由の起動	できる	できない

	disable	mask
稼働中のプロセス	止まらない	止まらない
解除の方法	enable	unmask (enable では戻らない)

● **start** さえ拒否する唯一の手段が **mask** です。`disable --now` は「自動起動を切って、いま止める」までで、その後の `systemctl start` は止められません。

● **mask** を解くのは **unmask** だけです。`mask` されたユニットに `enable` を打ってもエラーになり、`daemon-reload` も `mask` を解きません。まず `unmask`、それから必要なら `enable` や `start` です。

ユニットファイルを編集したら `daemon-reload` です。`systemd` は定義をメモリに持っているので、書き換えただけでは古い定義のまま動きます。`restart` はユニットファイルを読み直さず、`daemon-reload` はサービスを再起動しません(両方要るなら2つとも打つ)。

1-5-9 一覧と調査:何を見ているのかを取り違えない

コマンド	見ている対象	主な列
<code>systemctl list-units</code>	メモリ上にロード済みのユニット	UNIT / LOAD / ACTIVE / SUB / DESCRIPTION
<code>systemctl list-unit-files</code>	ディスク上に存在する全ユニットファイル	UNIT FILE / STATE (enabled/disabled など) / VENDOR PRESET

● **list-units** は実行時の状態、**list-unit-files** は設定ファイルの導入状態です。役割を入れ替えた選択肢が定番です(どちらも `service•target•mount•`

timer など複数の種類を扱う)。

```
例 systemctl list-units --type=service --state=active
```

`list-units` は既定で、**active なユニットだけ**を出します (停止中の `inactive` のユニットは出ません)。停止中のものも見たいときは `--all` を付け、`--state=failed` のように状態を指定して絞ることもできます。`status` には `--type` や `--state` の絞り込みはありません。

ユニットの定義そのものを読むなら `systemctl cat` です (本体とドロップインをまとめて表示)。`show` はプロパティの一覧です。`status` は実行状態と直近のログです。`list-dependencies` は依存のツリーです。どれもファイルの中身ではありません。

`systemctl status` の読み方も押さえます。

```
Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled; vendor
preset: enabled)
Active: active (running) since ...
```

- **Loaded:** 行: ユニットファイルが**読み込めたか** (`loaded` / `not-found` / `masked`) と、その**場所、自動起動の設定** (`enabled` / `disabled` / `static`)
- **Active:** 行: **いまの実行状態** (`active (running)` / `inactive (dead)` / `failed`)

この例は「ユニットファイルは `/usr/lib/systemd/system/` にあり、自動起動が有効で、いま稼働中」と読みます。`Loaded: loaded` は読み込みの成功です。`enabled` は自動起動の設定です (フォアグラウンド実行とは無関係)。

1-5-10 既定ターゲットの変更と、いますぐの切り替え

ここも「いま」と「次回」の対比です。

コマンド	効くタイミング	何が起きるか
<code>systemctl set-default <ターゲット></code>	次回以降の起動	<code>default.target</code> のシンボリックリンクを書き換える。いまの状態は変わらない
<code>systemctl isolate <ターゲット></code>	いますぐ	指定ターゲットとその依存ユニットだけを起動し、 含まれない他のユニットを停止する 。既定の設定は変わらない
<code>systemctl start <ターゲット></code>	いますぐ	指定ターゲットを起動キューへ入れるだけ。 他のユニットを停止しない ので、切り替えにはならない
<code>systemctl get-default</code>	—	現在の既定ターゲットを表示する

 **切り替えは `isolate` で、`start` は切り替えになりません。** `start rescue.target` では他のユニットが止まらず、稼働中のサービスが残ります。

 `init 1` や `telinit 1` でもレスキュー相当へ移れますが、SysV init 由来の書き方です(「`systemctl` を使って」という条件は満たさない)。`enable` は既定ターゲットを変えません。ユニット自身の `[Install]` に従ってリンクを作るだけです。

1-5-11 systemd-analyze :起動にかかった時間を調べる

systemd-analyze :起動時間とユニットの検査

書式 systemd-analyze [オプション] [<サブコマンド>] [<引数>...]

サブコマンド	何を出すか
time (既定)	カーネル・initrd・ユーザー空間それぞれに要した時間の合計
blame	ユニットごとの初期化時間を、長い順に一覧表示する
critical-chain	起動完了までの依存関係上のクリティカルパスをツリー表示する
dot	依存関係を GraphViz 形式のグラフとして出力する
dump	サービスマネージャの状態を人間可読な形で丸ごと出力する
verify	ユニットファイルを検査し、依存先の欠落や不正な指定を報告する

- 例 systemd-analyze blame
- 例 systemd-analyze critical-chain
- 例 systemd-analyze verify /etc/systemd/system/myapp.service

blame は「どのユニットが何秒かかったか」を遅い順に並べます。**critical-chain** は「起動完了までの依存の連鎖」をツリーで見せます。起動時間短縮のボトルネックになる遅いユニットを特定するなら前者、待ち合わせの構造を見

るなら後者です。`systemctl list-units` や `journalctl -b -p err` では、所要時間は出ません。

1-5-12 SysV init:systemd の前の世代を、いまでも問われる形で

SysV init では、PID 1 の `/sbin/init` が `/etc/inittab` を読み、ランレベルごとのスクリプトを番号順に 1 つずつ直列に実行します。途中で 1 つ詰まると、後ろが全部待たされます (systemd は依存を解いて並列に起動する)。

場所	何が置かれているか
<code>/etc/inittab</code>	既定のランレベルと、各ランレベルで何を 実行するかを書いた設定ファイル
<code>/etc/init.d/</code>	サービスごとの起動スクリプトの実体 (<code>start</code> ・ <code>stop</code> ・ <code>restart</code> ・ <code>status</code> を 引数に取る)
<code>/etc/rc.d/rc<N>.d/</code> (または <code>/etc/rc<N>.d/</code>)	ランレベル <code>N</code> のときに実行されるシンボリック リンクの置き場。実体は <code>/etc/init.d/</code> を指す

⚠ Red Hat 系は、`/etc/rc.d/` の下に `rc0.d` ～ `rc6.d` と `rc.local` を置きます (`/etc/rc<N>.d` はそこへのリンク)。Debian 系は `/etc/rc<N>.d/` を直に置きます。どちらも同じものを指します (書き方の違いだけ)。

1-5-13 `/etc/inittab` の書式

`/etc/inittab` は 1 行 1 エントリで、コロンで区切った 4 つの欄でできています。

書式 `<id>:<runlevels>:<action>:<process>`

欄	何を書くか
id	エントリを識別する 1～4 文字の名前。 中身に意味は無く、他の行と重ならない
runlevels	そのエントリを適用するランレベルを 区切り記号なしで並べる (2345 なら 2・3・4・5)
action	いつ・どう実行するか (下の表)
process	実行するコマンド(action が initdefault のときは空)

action	何をするか
initdefault	既定のランレベルを決める。 process 欄は使わない。システム全体で 1 行だけ
sysinit	ランレベルに入る前に、システム初期化として 1 回だけ実行する
respawn	プロセスが終了したら 自動的に起動し直す (getty がこれ)
wait	実行し、 終わるまで次へ進まない
once	そのランレベルに入るとき 1 回だけ実行する(終了を待たない)
boot / bootwait	起動時に 1 回だけ実行する(bootwait は終了を待つ)
off	そのエントリを無効にする

実物の抜粋を載せます。

```
# /etc/inittab

id:3:initdefault:

si::sysinit:/etc/rc.d/rc.sysinit

l3:3:wait:/etc/rc.d/rc 3

ca::ctrlaltdel:/sbin/shutdown -t3 -r now

1:2345:respawn:/sbin/mingetty tty1
```

行	意味
id:3:initdefault:	既定のランレベルは 3 (テキストのマルチユーザー)。末尾の process 欄は空
si::sysinit:/etc/rc.d/rc.sysinit	ランレベルに入る前にシステム初期化スクリプトを実行する。runlevels 欄も空でよい
ca::ctrlaltdel:...	Ctrl + Alt + Del の動作を定義する

● 既定のランレベルは、initdefault の行の 2 番目の欄で決まります。id 欄 (1 番目) でも process 欄 (4 番目) でもありません。initdefault に 0 (停止) や 6 (再起動) を書いてはいけません (起動するたびに落ちるか、再起動し続ける)。

⚠ /etc/inittab を書き換えても、それだけでは反映されません。init に読み直させるには telinit q (q は reload の意味) を実行します。

1-5-14 /etc/rc<N>.d/ の S と K

/etc/rc.d/rc <N> は、/etc/rc.d/rc<N>.d/ のリンクを名前順に見ていきます (ランレベル N に入るとき)。K で始まるものは引数 stop で、S で始まるも

のは引数 `start` で実行します。

書式 `<S または K><2桁の実行順><サービス名> → ../init.d/<サービス名>`

例 `/etc/rc.d/rc3.d/S55sshd -> ../init.d/sshd`
例 `/etc/rc.d/rc3.d/K90network -> ../init.d/network`
例 `/etc/rc3.d/S20apache2 -> ../init.d/apache2`

記号	意味
S	そのランレベルで起動する (<code>start</code> を渡す)。S は Start
K	そのランレベルで停止する (<code>stop</code> を渡す)。K は Kill
2 桁の数字	実行順。小さい方が先。依存する側を後ろの番号にする

● **数字は優先度ではなく実行順です。**「数字が大きいほど優先される」は誤りです。また **S と K は大文字**で、小文字で始まるファイル (`README` など) は実行されません。

`/etc/init.d/` のスクリプトは、手でも呼べます (引数に `start` ・ `stop` など) 。
Red Hat 系では `service` コマンドが同じことをする包み紙です。

例 `/etc/init.d/sshd restart`
例 `service sshd restart`
例 `service --status-all`

⚠ SysV 形式のスクリプトしか無いソフトウェアも、systemd でそのまま扱えます。 `systemd-sysv-generator` が、`/etc/init.d/` のスクリプトから `.service` ユニットの起動時に自動生成するからです。書き直さなくても、`systemctl start` / `enable` が効きます (generator が `.service` を作る)。ただし `systemd v260` でこの互換は削除されたので、新しい `systemd` ではユニットファイルが要ります。

例 `systemctl start myapp`

例 `systemctl enable myapp`

生成されたユニットは、`/run/systemd/generator.late/` に置かれる一時的なものです。`/etc/systemd/system/` に同名のユニットを置けば、そちらが優先されます (1-5-3)。スクリプトを直したら、`systemctl daemon-reload` で生成し直します。依存関係は LSB ヘッダ (1-5-17) から決まるので、ヘッダが無いと順序が思いどおりになりません。

1-5-15 自動起動を設定する: `chkconfig` と `update-rc.d`

`rc<N>.d/` のリンクは手で作りません。**Red Hat 系** は `chkconfig`、**Debian 系** は `update-rc.d` に任せます。

`chkconfig` : SysV init サービスの自動起動を設定する (Red Hat 系)

書式 `chkconfig [--level <ランレベル>] <サービス名> <on|off|reset>`

`chkconfig --list [<サービス名>]`

`chkconfig --add | --del <サービス名>`

オプション	意味
<code>--list</code>	サービスとランレベルごとの on/off を一覧する (サービス名を省くと全件)
<code>--level <ランレベル></code>	対象のランレベルを指定する (35 のように区切りなしで並べる)
<code>--add</code>	サービスを管理対象に加え、スクリプトの指定に従ってリンクを作る
<code>--del</code>	サービスを管理対象から外し、リンクを削除する
<code>on / off</code>	指定したランレベルで 起動する／しない に設定する
<code>reset</code>	スクリプトの既定値へ戻す

例 `chkconfig --list`

例 `chkconfig --list sshd`

例 `chkconfig sshd on`

例 `chkconfig --level 35 httpd on`

例 `chkconfig --level 2345 httpd off`

例 `chkconfig --add myapp`

例 `chkconfig --del myapp`

`chkconfig --list httpd` の `3:on` は、**ランレベル 3 に入るときに httpd を起動する設定**です (いま動いているという意味ではありません)。`3:off` なら、そのランレベルでは起動しない設定です。

 **`--level` を省くと、既定のランレベル (多くは 2・3・4・5) にまとめて設定されます。**ランレベルは区切りなしで並べ、`35` と `53` は同じ意味です。次の書き

方も、どれも同じ結果になります。

- `--level=35`
- 後ろへ回した `chkconfig httpd on --level 35`
- フルパスの `/sbin/chkconfig` (統合された環境では `/usr/sbin/chkconfig`)

⚠ ただし、`--level 3,5` のようなコンマ区切りは受け付けられません。

update-rc.d : SysV init サービスの起動リンクを作る (Debian 系)

書式 `update-rc.d [-f] <サービス名> remove`
`update-rc.d <サービス名> defaults`
`update-rc.d <サービス名> defaults-disabled`
`update-rc.d <サービス名> <disable|enable> [S|I|3|4|5]`

指定	意味
<code>defaults</code>	既定の設定でリンクを作る (起動するランレベルに <code>S</code> 、それ以外に <code>K</code>)
<code>defaults-disabled</code>	<code>defaults</code> と同じ配置でリンクを作るが、最初から無効の状態にする
<code>remove</code>	そのサービスのリンクをすべて削除する。 <code>/etc/init.d/</code> のスクリプトが残っていると拒否されるので <code>-f</code> を添える
<code>enable</code> / <code>disable</code>	既存のリンクの <code>S</code> と <code>K</code> を入れ替えて有効・無効を切り替える
<code>-f</code>	スクリプトが残っていても強制的に処理する

例 `update-rc.d apache2 defaults`

例 `update-rc.d -f apache2 remove`

例 `update-rc.d apache2 disable`

⚠ 現行の書式は `update-rc.d <名前> defaults` で、番号の引数は付けません。

起動と停止の順番は、LSB ヘッダ (1-5-17) の `Required-Start: / Required-Stop:` と `Default-Start: / Default-Stop:` から決まります。旧版の `update-rc.d apache2 defaults 20 80` (起動順・停止順の数字) は、Debian 6 以降は無視されます (依存関係ベースになったため)。フルパスなら `/usr/sbin/update-rc.d` です。

やりたいこと	Red Hat 系	Debian 系
自動起動を有効にする	<code>chkconfig <名前> on</code>	<code>update-rc.d <名前> defaults</code>
自動起動を無効にする	<code>chkconfig <名前> off</code>	<code>update-rc.d <名前> disable</code>
設定を一覧する	<code>chkconfig --list</code>	<code>ls /etc/rc*/d/</code>
リンクを消す	<code>chkconfig --del <名前></code> >	<code>update-rc.d -f <名前> remove</code>

1-5-16 `init` と `telinit` :ランレベルを切り替える

`init` / `telinit` :ランレベルの切り替えと設定の再読み込み

書式 `init <ランレベル|Q|q|S|s>`

`telinit <ランレベル|Q|q|S|s>`

引数	何が起こるか
0	システムを停止する(電源断)
1 または S・s	シングルユーザーモードへ移る
2 ~ 5	指定したマルチユーザーのランレベルへ移る
6	再起動する
q または Q	/etc/inittab を読み直す。ランレベルは変えない

```
例 init 1
例 telinit 5
例 telinit Q
```

telinit は init へのシンボリックリンクで、機能は同じです。systemd の環境でも互換のために残っています。

- init 3 は systemctl isolate multi-user.target に当たる
- init 5 は systemctl isolate graphical.target に当たる

いまのランレベルは、runlevel か who -r で確かめます (runlevel は前と現在の 2 つを表示し、前が無ければ N)。

1-5-17 LSB:ディストリビューションをまたぐ約束ごと

LSB (Linux Standard Base) は、配置・コマンド・起動スクリプトの書き方を決めた標準仕様です (ディストリビューションが違って同じように動かすた

め)。SysV init では、起動スクリプトの先頭のコメントブロックとして現れます。

```
#!/bin/sh

### BEGIN INIT INFO
# Provides:          myapp
# Required-Start:    $network $remote_fs
# Required-Stop:     $network $remote_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: My application daemon
# Description:       Starts and stops the myapp daemon.

### END INIT INFO
```

行	何を宣言しているか
### BEGIN INIT INFO / ### END INIT INFO	この間が LSB ヘッダであることを示す囲み
Provides:	このスクリプトが提供する名前(他のスクリプトから依存先として指せる)
Required-Start: / Required-Stop:	起動前・停止前に必要な依存先。 \$network・\$remote_fs・\$syslog など \$ 付きは LSB が定める仮想の名前
Default-Start:	update-rc.d <名前> defaults としたときに S リンクを作るランレベル
Default-Stop:	同じく K リンクを作るランレベル
Short-Description: / Description:	説明文

LSB のバージョンや、その OS が名乗っている配布名を調べるコマンドが `lsb_release` です。

```
例 lsb_release -a
例 lsb_release -i -r
```

1-5-18 `systemd-delta` : 上書きされた設定を洗い出す

`systemd` では、同じ名前のユニットファイルが `/etc` ・ `/run` ・ `/usr/lib` の 3 か所にあり得ます (1-5-3)。**どれが実際に効き、どれが隠れているか**をまとめて出すのが `systemd-delta` です。

`systemd-delta` : 上書き・マスクされた設定ファイルの一覧

```
書式 systemd-delta [--type=<種別>] [--diff=<yes|no>] [<接頭辞>...]
```

種別 (--type=)	何を出すか
<code>masked</code>	<code>/dev/null</code> へのリンクで 完全に無効化 されているユニット
<code>equivalent</code>	上書きしているが 中身が元と同じもの (消してよい候補)
<code>redirected</code>	別のファイルへのリンクになっているもの
<code>overridden</code>	<code>/etc</code> 側のファイルが <code>/usr/lib</code> 側を 上書きしているもの
<code>extended</code>	ドロップイン (<code>.d/* .conf</code>) で 追加・変更 されているもの

オプション	意味
<code>--diff=yes</code> / <code>--diff=no</code>	差分そのものを表示するかどうか(既定は表示する)
<code>--no-pager</code>	ページャに渡さずそのまま出力する

例 `systemd-delta --type=overridden`

例 `systemd-delta --type=extended --diff=no`

例 `systemd-delta systemd/system`

 **systemd-delta** は、ベンダーの既定とこの機械の実際の設定の差を見るコマンドです。「設定を変えたのに効かない」ときに、最初に打つコマンドです(どこで上書きされたかが分かる)。

1-5-19 ドロップインを作る・確かめる・消す

パッケージ提供のユニットは、丸ごとコピーして書き換えません。変えたい項目だけを、ドロップインに書きます：

操作	作られるファイル	備考
<code>systemctl edit <ユニット></code>	<code>/etc/systemd/system/<ユニット>.d/override.conf</code>	保存すると自動で <code>daemon-reload</code> が走る
<code>systemctl edit --runtime <ユニット></code>	<code>/run/systemd/system/<ユニット>.d/override.conf</code>	再起動で消える 。一時的な試しに向く
<code>systemctl edit --full <ユニット></code>	<code>/etc/systemd/system/<ユニット></code>	本体の丸ごとのコピー(ドロップインではない)

エディタには、**変えたい項目とそのセクション見出しだけ**を書きます。

`Restart=always` を足すなら、次の2行です：

```
[Service]
Restart=always
```

- 1つしか持てない項目 (`RestartSec=` など) : ドロップインの値が本体の値に**勝つ**。 `RestartSec=5` の本体に `RestartSec=30` を重ねると、有効なのは30秒
- 何度も書ける項目 (`ExecStart=` など) : ドロップインに書くと**追加**になる
- 追加でなく置き換えたいとき : 先に**空の** `ExecStart=` を書いて、一覧を空にする

効いている内容は、次のコマンドで確かめます：

```
$ systemctl cat myapp.service
# /etc/systemd/system/myapp.service
[Service]
RestartSec=5

# /etc/systemd/system/myapp.service.d/override.conf
[Service]
RestartSec=30

$ systemctl show -p FragmentPath -p DropInPaths -p RestartUsec
myapp.service
RestartUsec=30s
FragmentPath=/etc/systemd/system/myapp.service
DropInPaths=/etc/systemd/system/myapp.service.d/override.conf
```

- `systemctl cat` :ファイルごとに `# <パス>` の行を出し、本体とドロップインを続けて見せる
- `FragmentPath` :本体のファイルの場所
- `RestartUsec` :名前が `Usec` で終わる項目は、値が `30s` の形で出る
- 探索する場所の順: `systemctl show -p UnitPath <ユニット>` で見える。`/etc/systemd/system`、`/run/systemd/system`、`/usr/lib/systemd/system` の順

手でファイルを書き換えたら、`daemon-reload` が必要です。忘れて `restart` すると、次の警告が出て、古い値のまま動きます:

```
Warning: The unit file, source configuration file or drop-ins of
myapp.service changed on disk. Run 'systemctl daemon-reload' to reload
units.
```

`daemon-reload` の後の `restart` で、`RestartUsec` が新しい値に変わります。
`systemctl edit` で編集したときは、保存の時点で `daemon-reload` が済んでいます。

`systemctl revert <ユニット>` は、ドロップインと `/etc` の上書きを消して、ベンダー版に戻します。

- ドロップインだけがあるユニット: `override.conf` と `.d` ディレクトリが消える
- `/etc/systemd/system` にしか本体が無いユニット: 本体のファイルが残る

パッケージを更新しても新しい設定が反映されないときは、`/etc/systemd/system` に丸ごとのコピーが残っていないかを疑います。

- `FragmentPath` が `/etc/systemd/system/...` を指す
- `/usr/lib/systemd/system/...` に、新しい版もある
- この2つがそろえば、古いコピーが優先されている

`systemd-delta` の出力は、1行が「**[種別]** 効いているファイル → 隠されたファイル(またはドロップイン)」の形です:

```
[OVERRIDDEN] /etc/systemd/system/ssh.service →  
/usr/lib/systemd/system/ssh.service  
[EXTENDED] /usr/lib/systemd/system/user@.service →  
/usr/lib/systemd/system/user@.service.d/10-login-barrier.conf  
[EQUIVALENT] /etc/systemd/system/ctrl-alt-del.target →  
/usr/lib/systemd/system/ctrl-alt-del.target
```

- [OVERRIDDEN] :丸ごとの置き換え
- [EXTENDED] :ドロップインでの追加
- [EQUIVALENT] :実体と同じ (/etc 側が /usr/lib 側へのシンボリックリンクなど)。 /usr/lib のファイルをコピーして /etc に置いた場合は、内容が同じでも [OVERRIDDEN] と表示される (Debian 12 で確認)
- [MASKED] : /dev/null への置き換え

[OVERRIDDEN] が出たら、`systemctl revert` で戻し、必要な変更だけをドロップインに移します。

⚠ `/etc/systemd/system` に**実体のファイルがある名前**を `mask` すると、エラーになります:

```
Failed to mask unit: File /etc/systemd/system/myapp.service already  
exists.
```

`--force` を付けると、置き換えます。

1-5-20 is-enabled の値と、failed の読み方

`systemctl is-enabled` の答えは、次のとおりです:

値	意味
<code>enabled</code>	自動起動のリンクがある
<code>disabled</code>	リンクが無い(<code>[Install]</code> はある)
<code>static</code>	<code>[Install]</code> セクションが無い 。単独では <code>enable</code> も <code>disable</code> もできない。他のユニットの依存や、ソケットなどから起動される
<code>masked</code>	<code>/dev/null</code> へのリンクで止められている
<code>alias</code>	別名。 <code>ctrl-alt-del.target</code> は <code>reboot.target</code> への別名

`static` のユニットに `enable` を打つと、次のように表示されて、何も変わりません。

```
The unit files have no installation config (WantedBy=, RequiredBy=, Also=, Alias= settings in the [Install] section, and DefaultInstance= for template units). This means they are not meant to be enabled using systemctl.
```

`systemctl start` で起動することはできます。例えば `systemd-journald.service` は `static` です。

- `systemctl list-unit-files` の `STATE` 欄にも、同じ値が並ぶ
- `mask` した後: `is-enabled` は `masked` を返す。動いていれば `is-active` は `active` のまま(`mask` は動いているプロセスを止めない)

`systemctl --failed` は、失敗したユニットだけを一覧します。3つの列は、見ている層が違います：

列	意味
LOAD	ユニットの定義を読み込んだか (loaded ・ not-found ・ masked)
ACTIVE	大まかな状態 (active ・ inactive ・ failed)
SUB	ユニットの種類ごとの細かい状態 (running ・ dead ・ failed ・ exited など)

`loaded failed failed` は、「定義は読めた。起動を試みたら失敗して、そのまま止まっている」と読みます。

- 短い間に失敗が重なり、再起動の回数が上限 (`StartLimitBurst=`) を超える： `Result: start-limit-hit` で打ち切られる
- 原因の調べ方： `systemctl status` と `journalctl -u <ユニット>`
- 直したあと： `systemctl reset-failed` で `failed` の表示を消し、起動し直す

1-5-21 サービスの動かし方と再起動、依存の失敗

`[Service]` の `Type=` は、「いつ起動が完了したとみなすか」を決めます：

Type=	起動完了とみなす時点
simple (既定)	<code>ExecStart</code> のプロセスを起動した時点。 そのプロセスが主プロセス

Type=	起動完了とみなす時点
forking	親プロセスが終了した時点。デーモンが自分でバックグラウンドへ移る形。PIDFile= で主プロセスを示すことが多い
oneshot	ExecStart のプロセスが終了した時点。処理が終わったら止まる設定向け。RemainAfterExit=yes を足すと、終了後も active を保つ
notify	サービス自身が「準備ができた」と systemd に通知した時点

Restart= は、いつ再起動するかを決めます。

- **no** (既定) : 再起動しない
- **on-failure** : 終了コードが 0 以外、SIGKILL など異常なシグナルでの終了、タイムアウトのときだけ
- **always** : 止まったら必ず

on-failure では、**exit 0** で終わったプロセスと、**systemctl stop** で止めたプロセスは再起動されません。**RestartSec=** は、再起動までの待ち時間です：

```
[Service]
ExecStart=/usr/local/bin/worker
Restart=on-failure
RestartSec=2
```

この設定は、「**worker** が 0 以外で終了したら、2 秒待って再起動する」という意味です。再起動の回数は **systemctl show -p NRestarts <ユニット>** で読めま

す。

依存の指定は、**起動に失敗したときの扱い**が違います。`dep.service` が失敗する場合で比べます：

書き方	<code>dep.service</code> が失敗したとき
<code>Requires=dep.service</code> と <code>After=dep.service</code>	自分の起動が 取りやめられ 、 <code>inactive</code> のまま。A dependency job for <code>main.service</code> failed. と表示される。 <code>journalctl</code> には <code>Job main.service/start failed with result 'dependency'</code> と残る
<code>Wants=dep.service</code> と <code>After=dep.service</code>	自分は そのまま起動する
<code>Requires=dep.service</code> だけ (<code>After=</code> なし)	順序が無いので <code>dep.service</code> と 並行して起動 し、 <code>dep.service</code> が失敗しても自分は起動する (取りやめになるのは、 <code>After=</code> で順序を付けたとき)
<code>After=dep.service</code> だけ	順序だけを決める。 <code>systemctl start</code> しても <code>dep.service</code> は 起動されない

上の表の `dep.service` は、`Type=oneshot` で `ExecStart=/bin/false` のように、**起動の段階で失敗する**ものです。`Type=simple` (既定) だと、プロセスを起動できた時点で起動完了とみなすので、あとで終了しても `main.service` の起動は取りやめられません。

`Dependency failed` が出たら、自分ではなく**依存先の** `systemctl status` と `journalctl -u` を調べます。

`systemctl list-dependencies` で、逆向きにも調べられます。

- `systemctl list-dependencies <ユニット>` : 依存先の一覧
- `--reverse` を付ける: そのユニットを望んでいる側が出る。`enable` したユニットなら `multi-user.target` など
- `--all` を付ける: ターゲット以外のユニットも、再帰的に展開される

`systemctl isolate` は、`AllowIsolate=yes` のユニットにだけ使えます。

- `yes` のユニット: `multi-user.target` • `graphical.target` • `rescue.target` • `default.target`
- 普通のサービスに打つ: `Operation refused, unit may not be isolated.` で拒否される
- 確かめ方: `systemctl show -p AllowIsolate <ユニット>`

1-5-22 SysV の道具を systemd で打つとき

systemd の環境でも、昔のコマンドの多くは `systemctl` へのシンボリックリンクとして残っています。引数に応じて、ユニットの操作に翻訳されます。

- Debian 12 の `runlevel` • `telinit` • `shutdown` など: `/bin/systemctl` へのリンク
- `/sbin/init` : `systemd` 本体へのリンク

昔の書き方	systemd での動き
<code>telinit 0</code> / <code>init 0</code>	<code>systemctl poweroff</code>
<code>telinit 6</code> / <code>init 6</code> 、 <code>shutdown -r now</code>	<code>systemctl reboot</code> (<code>reboot.target</code>)
<code>telinit 1</code> (<code>s</code> ・ <code>S</code>)	<code>systemctl rescue</code>

昔の書き方	systemd での動き
<code>telinit 2 ~ 5</code>	<code>systemctl isolate</code> <code>runlevelN.target</code>
<code>telinit q (Q)</code>	<code>systemctl daemon-reload</code>
<code>telinit u (U)</code>	<code>systemctl daemon-reexec</code>

SysV init の `telinit q` は、「`/etc/inittab` を読み直す」指示でした。systemd は `/etc/inittab` を読まないの、`daemon-reload` に置き換わります：

SysV	systemd
<code>service <名> status</code>	<code>systemctl status <名></code>
<code>chkconfig <名> on / off</code>	<code>systemctl enable / disable <名></code>
<code>runlevel</code> (現在)	<code>systemctl get-default</code> (既定)・ <code>systemctl list-units --</code> <code>type=target</code> (現在)
<code>shutdown -h now</code>	<code>systemctl poweroff</code>

`runlevel` の出力は「直前 現在」の 2 つです。

- ランレベル 5 から `telinit 3` で移った直後： 5 3
- 起動直後など、直前が無いとき：左側が N (N 3)
- systemd の環境：現在のターゲットに対応する値を返す。直前が無ければ N (Debian 12・Rocky 9 のコンテナでは N 5)

SysV init の `/etc/inittab` の続き

- `initdefault` の行が無い SysV init：起動時にコンソールでランレベルを尋ねる

- **systemd**: `/etc/inittab` を読まない。同じ役割は `systemctl set-default multi-user.target` (ランレベル 3 相当) などが担う
- **wait**: そのランレベルに入るときに 1 回実行し、**終了まで待つ**
- **respawn**: 終了したら**すぐ起動し直す**。短時間に繰り返し終了すると、`init` は一定時間、起動を見合わせる
- `sysinit`・`boot`・`bootwait`: `runlevels` 欄は**無視**される。`sysinit` は `boot`・`bootwait` より先に実行される
- **ctrlaltdel**: `Ctrl + Alt + Del` で `init` が `SIGINT` を受け取ったときに実行する行。例は `shutdown -t3 -r now` (配布元で引数が違う)。この行を消すと、キー操作に反応しない

`rc<N>.d` の実行順

ランレベルを切り替えると、`/etc/rc<N>.d/` の中のリンクが、次の順に実行されます。

1. `K` のリンクが、**番号の小さい順に** `stop` で
2. `S` のリンクが、**番号の小さい順に** `start` で

```
$ ls -l /etc/rc5.d
K01cups -> ../init.d/cups
S01ssh  -> ../init.d/ssh
S02cron -> ../init.d/cron
```

ランレベル 5 に入るとき、`ssh` と `cron` は `start` で起動されます。動いていれば、`cups` は `stop` で止められます。`K` のリンクは「止める指示」であり、「何もしない」ではありません。停止の番号は、起動の番号の**逆順**になるように付けるのが慣例です。

- `update-rc.d ssh disable` : `rc2.d` ~ `rc5.d` の `S01ssh` を `K01ssh` に置き換える。リンクが残るので、`enable` で `S01ssh` に戻せる (Debian 12 で確認)
- `update-rc.d -f ssh remove` : リンクを消す
- `update-rc.d ssh defaults` : LSB ヘッダからリンクを作り直す
- `Default-Stop` : が空のスクリプト: 停止側の `K` リンクが作られない

Red Hat 系では、スクリプトの先頭に `# chkconfig: <起動するランレベル> <起動の番号> <停止の番号>` の行を書きます:

```
#!/bin/sh
# chkconfig: 2345 90 10
# description: legacy test daemon
```

`chkconfig --add legacyd` の結果は、次のとおりです (Rocky 9 で確認)。

- ランレベル 2・3・4・5: `S90legacyd` ができる
- それ以外 (0・1・6): `K10legacyd` ができる
- ランレベルの欄を `-` にする: 既定では、どのランレベルでも起動しない

`chkconfig --level 35 legacyd off` は、3 と 5 だけを変えます。2 と 4 は変わりません。設定を変えるだけで、いまは起動も停止もしません。`chkconfig --list` には、**SysV のサービスしか出ない**旨の注記が付きます。`systemd` のサービスは `systemctl list-unit-files` で調べます。

`systemd-sysv-generator` が作るユニット

`/etc/init.d/legacyd` だけのソフトも、`daemon-reload` の時点でユニットが作られます:

```
$ systemctl status legacyd
```

```
○ legacyd.service - LSB: legacy test
```

```
Loaded: loaded (/etc/init.d/legacyd; generated)
```

```
Active: inactive (dead)
```

Loaded: に generated と 出 ます。作 ら れ た 先 は /run/systemd/generator.late/legacyd.service です。中身は、次のとおりです。

- Type=forking、RemainAfterExit=yes、SuccessExitStatus=5 6
- ExecStart=/etc/init.d/legacyd start、ExecStop= は stop
- LSB ヘッダの Required-Start: : After= に変換される (\$remote_fs は remote-fs.target)

調整したいときは、/etc/systemd/system/legacyd.service.d/ にドロップインを置きます。/run の生成物は、直接編集しません。再起動や daemon-reload で作り直されるためです。

/etc/init.d/ のスクリプトは、引数に対応する分だけ動きます。

- 対応しない引数: Usage: /etc/init.d/legacyd {start|stop|status} のような使い方を出して終了する
- service ssh restart : /etc/init.d/ssh restart と同じスクリプトを、同じ引数で呼ぶ
- status : 終了コードでも状態を返す。止まっているときは 3

1-6 代替ブートローダー [202.3]

GRUB 2 以外にも、媒体や環境ごとのブートローダーがあります。副目標 202.3 の問いの中心は、「どの媒体・どの環境を担当するローダーか」です。

1-6-1 LILO:GRUB との対比だけを押さえる

⚠ **LILO は Version 4.5 の 202.3 からは外れています** (202.3 は SYSLINUX・ISOLINUX・PXELINUX と PXE、systemd-boot、U-Boot)。**LILO (Linux LOader)** は GRUB より前の BIOS 向けローダーです。**ファイルシステムを理解しません**。カーネルの位置を**マップファイル(既定は /boot/map)** にセクタ番号で記録し、起動時はそのセクタを直接読みます。そのため `/etc/lilo.conf` を書き換えたら、**必ず lilo を実行し直します** (MBR とマップの書き直し)。ファイルシステムを読める GRUB に、この再実行は要りません。

1-6-2 SYSLINUX プロジェクトの 4 種類

SYSLINUX は、媒体・ファイルシステムごとに 4 つのローダーを用意するプロジェクトです。どれがどの媒体かがそのまま問われます。

ローダー	対象	設定ファイル	典型的な用途
syslinux	FAT (FAT12/16/32)	<code>syslinux.cfg</code>	FAT の USB メモリ、フロッピー
extlinux	ext2/3/4・btrfs・XFS など	<code>extlinux.conf</code>	Linux のファイルシステム上のカーネルを起動する
isolinux	ISO9660	<code>isolinux.cfg</code>	CD・DVD などの光学メディア、ISO イメージ
pxelinux	ネットワーク (PXE)	<code>pxelinux.cfg/</code> 配下のファイル	TFTP でカーネルを取るネットワーク起動

名前が由来です (`ext linux` は `ext` 系、`iso linux` は `ISO9660`、`pxe linux` は `PXE`。残る素の `syslinux` が `FAT`)。設定ファイルの書式は 4 つとも共通で、

違うのはファイル名だけです。ISOLINUX は、`/isolinux/` か `/boot/isolinux/` の `isolinux.cfg` を読みます (`syslinux.cfg` があっても `isolinux.cfg` が優先)。

1-6-3 SYSLINUX 系の設定ファイルの書き方

```
DEFAULT linux
PROMPT 1
TIMEOUT 100
DISPLAY boot.msg

LABEL linux
    MENU LABEL Install Linux
    KERNEL vmlinuz
    APPEND initrd=initrd.img root=/dev/sda1 quiet

LABEL rescue
    KERNEL vmlinuz
    APPEND initrd=initrd.img rescue
```

指示語	何をするか
DEFAULT	時間切れのときに起動するラベル
PROMPT	1 なら boot: プロンプトを出して入力を待つ。0 なら出さない
TIMEOUT	入力を待つ時間。単位は 1/10 秒。0 を指定すると時間切れによる自動起動をしない

指示語	何をするか
<code>LABEL</code>	エントリの名前。プロンプトでこの名前を入力すると選ばれる
<code>KERNEL</code>	読み込むカーネルのファイル
<code>APPEND</code>	カーネルへ渡すパラメータ。 initrd もここで <code>initrd=</code> として指定する
<code>MENU LABEL</code>	メニュー画面での表示名
<code>DISPLAY</code>	起動時に表示するテキストファイル
<code>INCLUDE</code>	別の設定ファイルを取り込む

● **TIMEOUT 100 は 10 秒**です(100 秒と読ませる肢が定番)。再試行の回数でも、設定ファイルの大きさの上限でもありません。GRUB 2 は `linux` 行と `initrd` 行に分けます(1-3-7)。SYSLINUX 系は、**KERNEL 行でカーネルを、APPEND 行の `initrd=` で `initrd` を指定します。**

1-6-4 `syslinux` と `extlinux` :導入コマンドの違い

書式 `syslinux [-il--install] <デバイス>`

書式 `extlinux [-il--install] <ディレクトリ>`

例 `syslinux --install /dev/sdb1`

例 `syslinux -i /dev/sdb1`

例 `extlinux -i /boot`

● **`syslinux` はパーティションのデバイス名 (`/dev/sdb1`) を取り、`extlinux` はローダーを置くディレクトリ (`/mnt/boot`) を取ります。** ext4 の USB メモリ

なら `extlinux --install /mnt/boot`、FAT なら `syslinux --install /dev/sdb1` です。`-i` (`--install`) は、既存のブートセクタを上書きして導入します。

`isolinux` と `pxelinux` には導入コマンドがありません (`isolinux --install` や `pxelinux --install` は存在しない)。

- ISOLINUX:ISO イメージを作るときに `isolinux.bin` を埋め込む (`mkisofs` / `genisoimage` / `xorriso` に渡す)
- PXELINUX:TFTP サーバのルート配下に `pxelinux.0` を置くだけ

1-6-5 ISOLINUX と El Torito・ハイブリッド ISO

光学メディアからの起動は **El Torito** という規格で、二つの部品でできています。

部品	役割	ISOLINUX での実体
ブートイメージ	ファームウェアが実際に読み込んで実行するファイル	<code>isolinux.bin</code>
ブートカタログ	ブートイメージがイメージ内のどこにあるかを記録した索引	<code>boot.cat</code> (イメージ作成時に自動生成される)

● 実行されるローダー本体が `isolinux.bin`、その位置を記録する索引が `boot.cat` です (役割を入れ替えた肢が定番)。どちらもカーネルでも `initrd` でもありません。

`mkisofs` / `genisoimage` : 起動可能な ISO イメージを作る

書式 `mkisofs` [オプション] `-o` <出力.iso> <収録するディレクトリ>

オプション	意味
<code>-o <ファイル></code>	出力する ISO イメージのファイル名
<code>-b <パス></code>	El Torito のブートイメージ (ISOLINUX なら <code>isolinux/isolinux.bin</code>)
<code>-c <パス></code>	ブートカタログとして生成するファイル (<code>isolinux/boot.cat</code>)
<code>-no-emul-boot</code>	フロッピーやハードディスクの模倣(エミュレーション)を行わない
<code>-J</code>	Joliet 拡張 (Windows 向けの長いファイル名)
<code>-R / -r</code>	Rock Ridge 拡張 (UNIX のパーミッションや長いファイル名)

例 `mkisofs -o boot.iso -b isolinux/isolinux.bin -c isolinux/boot.cat -no-emul-boot -boot-load-size 4 -boot-info-table -J -R iso/`

ハイブリッド ISOは、光学メディアとしても、**USB メモリ(MBR を持つブロックデバイス)**としても起動できる ISO イメージです。`isohdpx.bin` は、そのとき **ISO の先頭 512 バイトへ書く MBR のテンプレート**です。`isohybrid` で後から加工するか、作成時に `xorriso -as mkisofs -isohybrid-mbr /usr/lib/syslinux/isohdpx.bin ...` と指定します。⚠ UEFI 用の FAT イメージ(`efiboot.img`。El Torito に別途登録する)とは別物で、署名や圧縮とも関係しません。

1-6-6 PXE ブート:DHCP と TFTP でネットワークから起動する

PXE (Preboot eXecution Environment) は、**DHCP と TFTP** を使ってネットワーク越しに起動する仕組みです。BIOS 環境の流れは次の 4 段です。

(1) クライアントの NIC が DHCP を要求する

↓ DHCP サーバが IP アドレスに加えて

next-server (ファイルを取りに行くサーバ) と

filename (取得するファイル名 = pxelinux.0) を返す

(2) クライアントが TFTP で pxelinux.0 を取得して実行する

↓

(3) PXELINUX が pxelinux.cfg/ 配下の設定ファイルを探して読む

↓

(4) 設定に書かれたカーネルと initrd を TFTP で取得して起動する

設定項目	何を通知するか
next-server	ファイルを取りに行くサーバ(TFTP サーバ)のアドレス
filename	取得するファイル(ブートファイル)の名前 (BIOS 環境なら pxelinux.0)

 **next-server** は **TFTP サーバ**です (DNS・ゲートウェイ・NFS サーバではない。ゲートウェイはオプション 3 の **routers**)。 **filename** は、**最初に取りに行くローダーのファイル**です。ルートファイルシステムのパスでも、initramfs でも、 **grub.cfg** でもありません。

ISC **dhcpd** では、 **next-server** が DHCP ヘッダの **siaddr** を、 **filename** が **file** フィールドを設定します。同じ役割のオプションが **66 (TFTP server**

name)と 67 (Bootfile name)です (`sname` ・ `file` がオプションの格納に転用されたときに代わりを務める)。

```
subnet 192.168.10.0 netmask 255.255.255.0 {
    range 192.168.10.100 192.168.10.200;
    option routers 192.168.10.1;
    next-server 192.168.10.5;
    filename "pxelinux.0";
}
```

BIOS の PXELINUX 起動に要るのは、次の 3 つです。

- `next-server` と `filename` を通知する DHCP サーバ
- TFTP ルート配下の `pxelinux.0`
- `pxelinux.cfg/` 配下の設定ファイル

ESP の `grubx64.efi` (UEFI の部品)、MBR の GRUB、NFS エクスポートは、起動の工程そのものには要りません (NFS はルートをネットワークに置くディスクレス構成で使う)。

1-6-7 `pxelinux.cfg/` の探索順

PXELINUX は、**TFTP ルート配下の `pxelinux.cfg` ディレクトリ**から設定ファイルを探します。

名前の形	由来	例
UUID	DHCP オプション 97 で渡されたクライアントの UUID	b8945908-d6a6-41a9-611d-74a6ab80b83d

名前の形	由来	例
MAC アドレス	ARP 種別 (01-) を頭に付け、区切りをハイフンにした小文字表記	01-88-99-aa-bb-cc-dd
IP アドレス	IP アドレスを 大文字 16 進 で表したもの。 1 桁ずつ削りながら探す	C000025B → C000025 → … → C
default	どれも見つからなかったときに最後に読まれる	default

例 /var/lib/tftpboot/pxelinux.cfg/01-88-99-aa-bb-cc-dd

例 /var/lib/tftpboot/pxelinux.cfg/default

● 探索は **UUID → MAC アドレス → IP の 16 進 (1 桁ずつ削る) → default** の順です。**default は最後**です。ホスト名やドメイン名は使いません。端末ごとに違う設定を配るなら、MAC アドレスの名前でファイルを置きます。

⚠ DHCP の拡張オプションで起点を指定することもでき、配られていれば標準の探索より先に使われます。

- **209** : pxelinux.configfile (設定ファイル名)
- **210** : pxelinux.pathprefix (パスの接頭辞)
- **211** : pxelinux.reboottime (TFTP 失敗時の再起動までの秒数)

pxelinux.0 はローダー本体のファイルです。/var/lib/tftpboot は TFTP のルートです。設定ファイルを探すのは、その配下の pxelinux.cfg です。

1-6-8 UEFI 環境でのネットワーク起動

UEFI でも **DHCP と TFTP の流れは同じ**ですが、ファームウェアが実行できるのは PE 形式の EFI アプリケーションだけです。

	BIOS 環境	UEFI 環境
DHCP で通知するファイル	pxelinux.0	shim.efi や grubx64.efi (BOOTX64.EFI)
PXELINUX 相当	pxelinux.0	syslinux.efi

 **pxelinux.0** は BIOS のリアルモード用で、UEFI は変換しても実行できません。BIOS と UEFI が混在する環境では、**DHCP サーバがクライアントのアーキテクチャ種別(オプション 93)を見て、配るファイルを切り替えます**。この値は Client System Architecture と呼ばれ、IANA の登録では 0 が BIOS の x86、7 が UEFI の x64 です。

1-6-9 systemd-boot:UEFI 専用の軽量ブートマネージャ

systemd-boot (旧称 gummiboot) は、**UEFI 専用**の軽量なブートマネージャです。**EFI スタブを持つカーネル**は、それ自体を EFI アプリケーションとしてファームウェアが読めます。そのため systemd-boot は、メニューを出して選ばれたカーネルを起動させるだけです(ファイルシステムを読んでカーネルを載せる重い仕事はしない)。設定はすべて **ESP の上**に置きます。

ファイル	何を書くか
loader/loader.conf	全体の設定(既定エントリ default、待ち時間 timeout、エディタの可否 editor)

ファイル	何を書くか
loader/entries/<名前>.conf	起動項目 1 つぶん。title・linux・ initrd・options・version・ machine-id の各行

```
title    Fedora Linux
version  6.5.6-300.fc39.x86_64
linux    /vmlinuz-6.5.6-300.fc39.x86_64
initrd   /initramfs-6.5.6-300.fc39.x86_64.img
options  root=UUID=abcd-1234 ro quiet
```

bootctl : systemd-boot の導入と管理

サブコマンド	何をするか
install	systemd-boot を ESP へ導入し、UEFI のブートエントリを登録する
update	導入済みの systemd-boot を新しい版へ入れ替える
remove	ESP とファームウェアの一覧から systemd-boot を取り除く
status	ファームウェアとブートローダーの現在の状態を表示する
list	利用できる起動項目を一覧する
set-default <ID>	既定の起動項目を恒久的に設定する

```
例 bootctl install
例 bootctl --path=/boot/efi install
例 bootctl status
```

ESP の位置は `/efi`・`/boot`・`/boot/efi` から自動判別されます。明示するなら `--esp-path=` (旧称 `--path=`) を付けます (サブコマンドの前でも後でもよい)。● **初回の導入は `install` です。**`update` は導入済みのものを新しくするだけです (`grub-install` は GRUB 用)。**systemd-boot は MBR を使いません。** El Torito (ISOLINUX の仕事) や TFTP による起動 (PXELINUX の仕事) もしません。

1-6-10 U-Boot:組込み機器のブートローダー

U-Boot (Das U-Boot) は、**ARM・PowerPC・RISC-V など組込み機器で広く使われるブートローダー**です。

特徴	中身
対象	組込み機器 (ARM・PowerPC・RISC-V など)。x86 の BIOS/UEFI を置き換えるものではない
操作	対話的なコマンドプロンプト を持ち、環境変数で挙動を決める
起動元	SD カード・eMMC・NAND/NOR フラッシュ・ネットワーク (TFTP) など多様な媒体
カーネルの形式	<code>mkimage</code> でヘッダを付けた <code>ulimage</code> 、または FIT イメージ
起動コマンド	<code>bootm (ulimage)・bootz (zImage)・booti (Image)</code>

特徴	中身
起動スクリプト	<code>boot.scr</code> (<code>mkimage</code> で作る)・ <code>uEnv.txt</code> (環境変数をテキストで書く)

```
例  mkimage -A arm -O linux -T kernel -C none -a 0x80008000 -e
    0x80008000 -n "Linux" -d zImage uImage

例  mkimage -A arm -O linux -T script -C none -n "boot script" -d
    boot.cmd boot.scr
```

⚠ U-Boot は PC 向けの BIOS 実装でも、ESP のローダーを読む UEFI の代替でも、ネットワーク起動専用でもありません(多様な媒体から起動できる汎用のローダー)。

1-7 章末チェック [202.1/202.2/202.3]

混同しやすい対の概念

対	違い
BIOS レガシーブートと UEFI ブート	ディスク先頭 512 バイトのコードを読むか、ESP 上の <code>.efi</code> ファイルを読むか
MBR と ESP	ディスク先頭 512 バイトの領域 (パーティションではない) か、FAT32 の通常のパーティションか
ESP と BIOS Boot Partition	UEFI 用で FAT32・ <code>/boot/efi</code> にマウントするか、GPT+BIOS 用でファイルシステムを持たず数百 KB〜1 MB か

対	違い
<code>/boot/efi/</code> と <code>/boot/grub2/</code>	ESP (<code>.efi</code> を置く) か、 <code>grub.cfg</code> と GRUB モジュールを置く通常のディレクトリか
<code>grub-install</code> と <code>grub-mkconfig</code>	ブートローダー 本体 を配置するか、設定ファイル <code>grub.cfg</code> を 生成 するか
<code>grub2-mkconfig</code> と <code>update-grub</code>	Red Hat 系の素のコマンド (<code>-o</code> が必須) か、Debian 系のラッパー (引数なし 1 語) か
<code>grub2-mkconfig</code> の <code>-o</code> の有無	ファイルへ書き込むか、標準出力に表示して終わるか
GRUB Legacy の <code>root (hd0,0)</code> と GRUB 2 の <code>set root=(hd0,1)</code>	パーティション番号が 0 起点か 1 起点か (ディスク側はどちらも 0 起点)
<code>grub.cfg</code> の直接編集と生成元の編集	再生成で 上書きされて消える か、残るか
メニューの <code>e</code> 編集と <code>/etc/default/grub</code>	その回の起動だけ効くか、次回以降も効くか
<code>grub></code> と <code>grub rescue></code>	通常モードに入れているか、最小コマンドしか使えないか
UEFI シェルと <code>grub rescue></code> と <code>initramfs</code> のシェル	ファームウェアか、ブートローダーか、カーネル起動後・ルート切り替え前か
<code>efibootmgr -n</code> と <code>-o</code>	次回 1 回限り (<code>BootNext</code>) か、恒久的な順序 (<code>BootOrder</code>) か
<code>efibootmgr -o</code> と <code>-O</code>	<code>BootOrder</code> を設定するか、 <code>BootOrder</code> 変数を 削除 するか
<code>efibootmgr</code> 素と <code>-v</code>	ラベルの一覧までか、ローダーファイルのパスまでか

対	違い
<code>initrd</code> と <code>initramfs</code>	ブロックデバイスとして扱う旧方式か、 <code>cpio</code> を <code>tmpfs</code> へ展開する新方式か
<code>dracut</code> と <code>update-initramfs</code>	Red Hat 系か、Debian 系か
<code>update-grub</code> と <code>update-initramfs</code>	GRUB の 設定ファイル を作り直すか、 <code>initramfs</code> を作り直すか
<code>/etc/systemd/system/</code> と <code>/usr/lib/systemd/system/</code>	管理者のローカル設定 (最優先) か、パッケージ提供の既定 (最下位) か
基本ユニットファイルとドロップイン	最初に見つかった 1 つだけが採用されるか、内容が重ねられるか
<code>Requires=</code> と <code>Wants=</code>	依存先の失敗が波及する強い依存か、波及しない弱い依存か
<code>Requires=</code> と <code>BindsTo=</code>	依存先の停止に連動しないか、連動して自分も停止するか
<code>Wants=</code> / <code>Requires=</code> と <code>After=</code> / <code>Before=</code>	一緒に起動されることを表すか、 順序 を表すか
<code>multi-user.target</code> と <code>graphical.target</code>	ランレベル 3 相当か、ランレベル 5 相当か
<code>rescue.target</code> と <code>emergency.target</code>	ローカルファイルシステムを マウントする か、 マウントしない か
<code>enable</code> と <code>start</code>	次回起動時のリンクを作るか、いますぐプロセスを起こすか
<code>disable</code> と <code>mask</code>	自動起動のリンクを消すだけか、 <code>/dev/null</code> へのリンクで 手動起動も封じる か
<code>mask</code> の解除	<code>enable</code> では戻らず、 <code>unmask</code> が必要

対	違い
<code>is-active</code> と <code>is-enabled</code>	いま稼働中かを返すか、次回自動起動する設定かを返すか
<code>list-units</code> と <code>list-unit-files</code>	ロード済みユニットの 実行時状態 か、ディスク上の全ユニットファイルの 導入状態 か
<code>systemctl cat</code> と <code>show</code> と <code>status</code>	定義ファイル(ドロップイン込み)の中身か、プロパティ一覧か、実行状態とログか
<code>isolate</code> と <code>start</code> (ターゲット)	含まれない他ユニットを 停止して切り替える か、起動キューに足すだけか
<code>isolate</code> と <code>set-default</code>	いますぐ切り替えるか、次回起動時の既定を変えるか
<code>daemon-reload</code> と <code>restart</code>	ユニットファイルを 読み直す か、プロセスを 起こし直す か
<code>systemd-analyze blame</code> と <code>critical-chain</code>	ユニットごとの所要時間の一覧か、依存関係上のクリティカルパスのツリーか
SysV init と <code>systemd</code>	スクリプトを番号順に 直列 に実行するか、依存関係を解いて 並列 に起動するか
<code>/etc/inittab</code> の <code>id</code> 欄と <code>runlevels</code> 欄	重ならないければ何でもよい識別子か、 適用するランレベル か(既定は後者の <code>initdefault</code> 行)
<code>rc<N>.d/</code> の <code>S</code> と <code>K</code>	<code>start</code> を渡して起動するか、 <code>stop</code> を渡して停止するか
<code>chkconfig</code> と <code>update-rc.d</code>	Red Hat 系か、Debian 系か
<code>chkconfig <名前> on</code> と <code>--level 35 <名前> on</code>	既定のランレベルにまとめて設定するか、指定したランレベルだけに設定するか

対	違い
<code>update-rc.d defaults</code> と <code>remove</code>	LSB ヘッダに従ってリンクを作るか、リンクを全部消すか (<code>-f</code> が要る)
<code>init 3</code> と <code>telinit q</code>	ランレベルを切り替えるか、 <code>/etc/inittab</code> を読み直すだけか
LSB ヘッダの <code>Required-Start:</code> と <code>Default-Start:</code>	起動前に必要な依存先か、 <code>defaults</code> で <code>S</code> リンクを作るランレベルか
<code>systemd-delta</code> と <code>systemd-analyze</code>	ベンダー既定と実設定の差を出すか、 起動時間 を出すか
<code>/dev/nvme0n1p1</code> と <code>/dev/sda1</code>	ネームスペース番号の後ろに <code>p</code> が入る NVMe か、SATA/SCSI か
<code>chroot</code> 前の <code>--bind</code>	<code>/dev</code> ・ <code>/proc</code> ・ <code>/sys</code> を持ち込まないと <code>grub-install</code> が失敗する
LILO と GRUB	設定変更のたびに <code>lilo</code> の再実行が要るか、要らないか (LILO は V4.5 では範囲外)
<code>syslinux</code> と <code>extlinux</code> と <code>isolinux</code> と <code>pxelinux</code>	FAT か、ext 系/btrfs か、ISO9660 か、ネットワーク (PXE) か
<code>syslinux.cfg</code> と <code>extlinux.conf</code> と <code>isolinux.cfg</code> と <code>pxelinux.cfg/</code>	対応するローダーごとの設定ファイル名 (書式は共通)
<code>syslinux --install</code> と <code>extlinux --install</code>	デバイス名 を取るか、 ディレクトリ を取るか
<code>isolinux.bin</code> と <code>boot.cat</code>	読み込まれるブートイメージか、生成されるブートカタログか
<code>mkisofs</code> の <code>-b</code> と <code>-c</code>	ブートイメージの指定か、ブートカタログの指定か

対	違い
<code>isohdpx.bin</code> と <code>efiboot.img</code>	ISO 先頭に埋め込む MBR テンプレートか、 El Torito に登録する UEFI 用 FAT イメージ か
<code>TIMEOUT 100</code> の読み方	100 秒ではなく 10 秒 (単位は 1/10 秒)
<code>next-server</code> と <code>filename</code>	TFTP サーバのアドレスか、取得するファイル 名か
<code>pxelinux.0</code> と <code>pxelinux.cfg</code>	TFTP で最初に取りるローダー本体か、設定フ ァイルを置くディレクトリか
PXE 設定の探索順	UUID → MAC → IP の 16 進(1 桁ずつ削 る) → <code>default</code> (<code>default</code> は 最後)
BIOS の PXE と UEFI の PXE	<code>pxelinux.0</code> を配るか、 <code>shim.efi</code> / <code>grubx64.efi</code> を配るか
systemd-boot と GRUB	ESP 上の <code>.conf</code> で EFI スタブ付きカーネ ルを起動する UEFI 専用か、BIOS でも UEFI でも使える汎用か
<code>bootctl install</code> と <code>update</code>	初回の導入か、導入済みのものの入れ替え か
U-Boot と UEFI	組込み機器向けのブートローダーか、PC の ファームウェア仕様か

この章の必修コマンド一覧

コマンド	何をするか
<code>efibootmgr</code>	UEFI の NVRAM ブートエントリの一覧・作 成・削除・順序変更

コマンド	何をするか
<code>grub2-mkconfig</code> / <code>grub-mkconfig</code>	<code>/etc/default/grub</code> と <code>/etc/grub.d/</code> から <code>grub.cfg</code> を生成する
<code>update-grub</code>	Debian 系のラッパー (<code>grub-mkconfig -o /boot/grub/grub.cfg</code> 相当)
<code>grub2-install</code> / <code>grub-install</code>	GRUB 本体を MBR や ESP へ導入する
<code>ls</code> / <code>set</code> / <code>insmod</code> / <code>normal</code>	<code>grub rescue></code> の最小シェルから手動で復旧する
<code>dracut</code>	Red Hat 系で <code>initramfs</code> を生成する
<code>mkinitramfs</code>	Debian 系で <code>initramfs</code> を 1 個生成する
<code>update-initramfs</code>	Debian 系で <code>initramfs</code> をまとめて作成・更新・削除する
<code>systemctl</code>	ユニットの起動・停止・有効化・禁止・状態確認・ターゲット切り替え
<code>systemd-analyze</code>	起動時間の分析 (<code>blame</code> ・ <code>critical-chain</code>) とユニットファイルの検査
<code>service</code>	<code>/etc/init.d/</code> のスクリプトを名前呼び出す (<code>service <名前> restart</code>)
<code>runlevel</code> / <code>who -r</code>	いまのランレベルと 1 つ前のランレベルを表示する
<code>mount</code> / <code>chroot</code>	レスキュー起動から、ディスク上のシステムを本来のルートとして扱う
<code>chkconfig</code>	SysV init のサービスの自動起動を設定・確認する (Red Hat 系)

コマンド	何をするか
<code>update-rc.d</code>	SysV init のサービスの起動リンクを作成・削除する (Debian 系)
<code>init</code> / <code>telinit</code>	ランレベルを切り替える / <code>telinit q</code> で <code>/etc/inittab</code> を読み直す
<code>systemd-delta</code>	上書き・マスクされたユニットファイルの差分を一覧する
<code>syslinux</code>	FAT のパーティションへ SYSLINUX を導入する
<code>extlinux</code>	ext 系・btrfs のディレクトリへ EXTLINUX を導入する
<code>mkisofs</code> / <code>genisoimage</code>	ISOLINUX を埋め込んだ起動可能な ISO イメージを作る
<code>isohybrid</code>	ISO の先頭に <code>isohdpx.bin</code> の MBR を埋め、USB から起動できるようにする
<code>bootctl</code>	systemd-boot を ESP へ導入・更新・削除し、起動項目を管理する
<code>mkimage</code>	U-Boot 用の <code>uImage</code> や起動スクリプト <code>boot.scr</code> を作る

試験ポイント

- **MBR は 512 バイト＝ブートストラップコード 446 + パーティションテーブル 64 + シグネチャ 0x55AA 2。**カーネルや initramfs はここに入らない
- **ESP は FAT32 で、多くのディストリビューションが `/boot/efi` にマウントする。**GPT+BIOS レガシーで要るのは、ファイルシステムを持たない BIOS

Boot Partition (`bios_grub`) で、これは別物

- `grub.cfg` は生成物。直接編集しない理由は「再生成で上書きされて消えるから」(書けないからではない)
- `grub2-mkconfig` は `-o` が無いと標準出力に出るだけ。ファイルへ反映するには `-o /boot/grub2/grub.cfg`
- `grub2-install` の導入先はディスク全体 (`/dev/sda`)。 `/dev/sda1` のようなパーティション指定は誤り
- コマンド名の系統: 「2」付きの `grub2-mkconfig` は Red Hat 系、 `update-grub` は Debian 系だけ。Debian 系には互換用の `update-grub2` (`update-grub` を呼ぶだけのスタブ) も入っている
- カーネルパラメータの恒久化は 2 手セット。 `/etc/default/grub` の `GRUB_CMDLINE_LINUX` に書き、 `grub2-mkconfig` で再生成する。メニューの `e` 編集はその回限り
- パーティション番号は GRUB Legacy が 0 起点 (`(hd0,0)`)、GRUB 2 が 1 起点 (`(hd0,1)`)。ディスク側はどちらも 0 起点
- `grub rescue>` では Linux のコマンドは動かない。使えるのは `ls` ・ `set` ・ `insmod` ・ `normal` など GRUB 内蔵のものだけ
- `efibootmgr` の小文字 `-o` は BootOrder の設定、大文字 `-O` は BootOrder の削除。 `-n` は次回 1 回限り、 `-o` はカンマ区切りで恒久的な順序
- `initramfs` の役割は「本来のルートをマウントするためのモジュールと道具を一時的に提供し、 `switch_root` で切り替えること」。展開するのはファームウェアではなくカーネル
- ユニットファイルの優先順位は `/etc/systemd/system/` > `/run/systemd/system/` > `/usr/lib/systemd/system/` 。同名なら最初

に見つかった 1 つだけが採用され、マージされるのはドロップインの側

- **Requires= / Wants= は起動順序を保証しない。**順序は **After= / Before=**、依存先の停止への連動は **BindsTo=**
- **multi-user.target がランレベル 3、graphical.target がランレベル 5。**
rescue.target は 1
- **rescue.target はローカルファイルシステムをマウントし、emergency.target はマウントしない。**どちらもネットワークサービスは起動しない
- **完全に起動を禁止したいなら mask (/dev/null へのリンク)。****disable** は自動起動を切るだけで **start** は通る。解除は **unmask** であって **enable** ではない
- **enable は次回のリンクを作るだけ、start はいま起こすだけ。**両方したいなら **enable --now**
- **list-units はロード済みの実行時状態、list-unit-files はディスク上の全ユニットファイルの導入状態**
- **ユニットファイルを編集したら daemon-reload 。****restart** は読み直さないし、**daemon-reload** は再起動しない
- **ターゲットを「いま」切り替えるのは isolate (=他を停止する)、「次回」の既定を変えるのは set-default 。****start** は切り替えにならない
- **systemd-analyze blame はユニットごとの所要時間の一覧、critical-chain は依存の連鎖のツリー**
- **/etc/inittab の行は id:runlevels:action:process の 4 欄。**既定ランレベルは **action** が **initdefault** の行で決まり、**process** 欄は空
- **/etc/rc<N>.d/ のリンクは S が起動・K が停止、続く 2 桁が実行順。**実体は **/etc/init.d/** のスクリプト

- 自動起動の設定は Red Hat 系が `chkconfig`、Debian 系が `update-rc.d`。 `chkconfig --list` が一覧、`chkconfig <名前> on` が有効化
- `telinit q` は `/etc/inittab` を読み直すだけでランレベルを変えない
- LILO は Version 4.5 では範囲外。覚えるのは「設定変更のたびに `lilo` の再実行が要るのが LILO、要らないのが GRUB」の対比だけ
- SYSLINUX 系の担当: `syslinux` =FAT、`extlinux` =ext 系/btrfs、`isolinux` =ISO9660、`pxelinux` =ネットワーク。設定ファイルの書式は共通で、ファイル名だけが違う
- 導入コマンドの引数: `syslinux` はデバイス名、`extlinux` はディレクトリ。`isolinux` と `pxelinux` に独立した導入コマンドは無い
- `TIMEOUT` の単位は 1/10 秒 (`TIMEOUT 100` は 10 秒)。秒だと思って読むと 10 倍ずれる
- El Torito では `isolinux.bin` がブートイメージ (`-b`)、`boot.cat` がブートカタログ (`-c`)
- `isohdpx.bin` は ISO 先頭に埋める MBR のテンプレート。これで光学メディアと USB のどちらからも起動できるハイブリッド ISO になる
- PXE は DHCP の `next-server` (TFTP サーバ) と `filename` (`pxelinux.0`) + TFTP。設定の探索は UUID → MAC → IP の 16 進 → `default` の順で、`default` は最後
- UEFI の PXE で配るのは `shim.efi` / `grubx64.efi`。 `pxelinux.0` は BIOS のリアルモード用で UEFI では実行できない
- `systemd-boot` は UEFI 専用。 `bootctl install` で ESP へ導入し、`loader/loader.conf` と `loader/entries/*.conf` で設定する
- U-Boot は組み込み機器向け。 `mkimage` で作った `uImage` を `bootm` などで起動し、`boot.scr` / `uEnv.txt` で手順を書く

■ サンプルはここまでです

続き(第2章～巻末)は、LPIC-2 201 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

SAMPLE

LPIC-2 201試験 学習用テキスト(無料サンプル)

版

2026年10月版(LPI公式 LPIC-2 Exam 201 Objectives Version 4.5 準拠)

発行

IT資格道場(<https://www.shikaku-doujou.com>)

お問い合わせ

info@shikaku-doujou.com

LPI および LPIC は Linux Professional Institute Inc. の商標または登録商標です。Linux は Linus Torvalds 氏の米国およびその他の国における商標または登録商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式シラバスの内容を基に作成したもので、Linux Professional Institute との提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点は info@shikaku-doujou.com までご連絡ください。