

# Shell Scripting

2015/08/10

# BABAROT



@b4b4r07



@b4b4r07



# BABAROT



**BABAROT**  
b4b4r07

Tokyo, Japan  
b4b4r07@gmail.com

Contributions

Repositories

Public activity

Edit profile

## Popular repositories

[dotfiles](#)

dotfiles ❤ Testing my dotfiles repo on OS ...

47 ★

[gomi](#)

gomi is a simple trash tool that works on ...

29 ★

[enhanced](#)

A simple tool that provides enhanced cd c...

13 ★

[gch](#)

List the changes that are applied to the re...

8 ★

[vim-shellutils](#)

A simple UNIX Shell commands emulator ...

5 ★

## Repositories contributed to

[unicorn/awesome-zsh-plugins](#)

597 ★

Collection of ZSH frameworks, plugins & them...

[mattn/go-scan](#)

53 ★

[Homebrew/homebrew](#)

The missing package manager for OS X.

24,427 ★

[golang/crypto](#)

[mirror] Go supplementary cryptography libraries

163 ★

[tmux-plugins/tmux-online-status](#)

17 ★

Tmux plugin that displays online status of your...

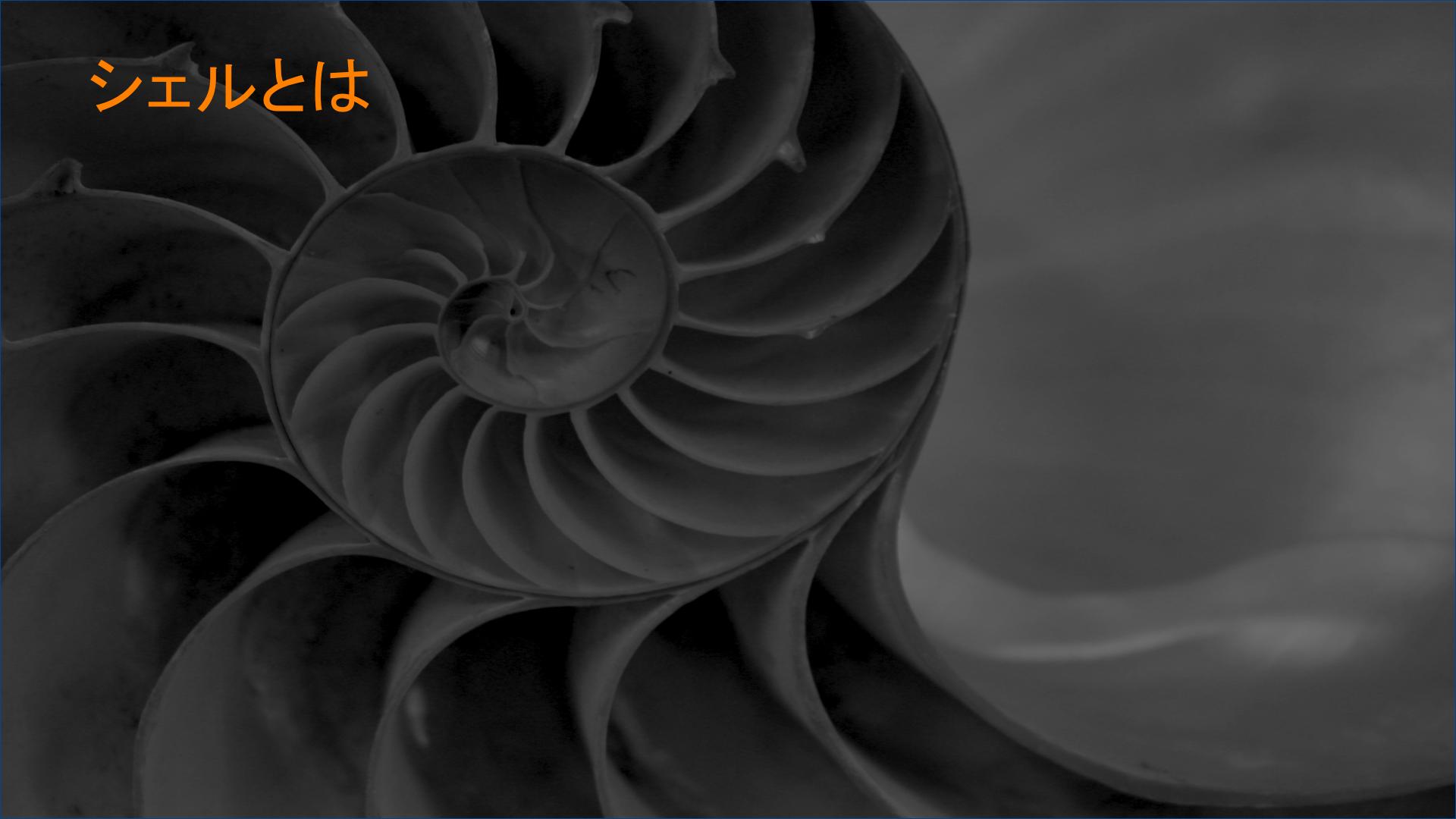


A simple-trash CLI tool.

Find Out More

**<http://b4b4r07.com/gomi>**

シェルとは





# シェルとは

- 大きく分けて 2 つある

# シェルとは

- 大きく分けて 2 つある
  - CLI
  - GUI

# シェルとは

- 大きく分けて 2 つある
  - CLI (コマンドラインインターフェースを提供する)
  - GUI (グラフィカルインターフェースを提供する)

# シェルとは

- ・ 大きく分けて 2 つある
  - CLI (コマンドラインインターフェースを提供する)
  - GUI (グラフィカルインターフェースを提供する)
- ・ どちらにせよ、共通して「**プログラムの起動**」を担う

# シェルとは

- ・ 大きく分けて 2 つある
  - CLI (コマンドラインインターフェースを提供する)
  - GUI (グラフィカルインターフェースを提供する)
- ・ どちらにせよ、共通して「**プログラムの起動**」を担う
- ・ 今回扱うのは、CLI のシェル

# シェルとは

- ・ 大きく分けて 2 つある
  - CLI (コマンドラインインターフェースを提供する)
  - GUI (グラフィカルインターフェースを提供する)
- ・ どちらにせよ、共通して「**プログラムの起動**」を担う
- ・ 今回扱うのは、CLI のシェル
  - 中でも UNIX シェルと呼ばれるもの

# シェルとは

- ・ 大きく分けて 2 つある
  - CLI (コマンドラインインターフェースを提供する)
  - GUI (グラフィカルインターフェースを提供する)
- ・ どちらにせよ、共通して「**プログラムの起動**」を担う
- ・ 今回扱うのは、CLI のシェル
  - 中でも UNIX シェルと呼ばれるもの
  - MS-DOS やその他 OS のシェルも存在する

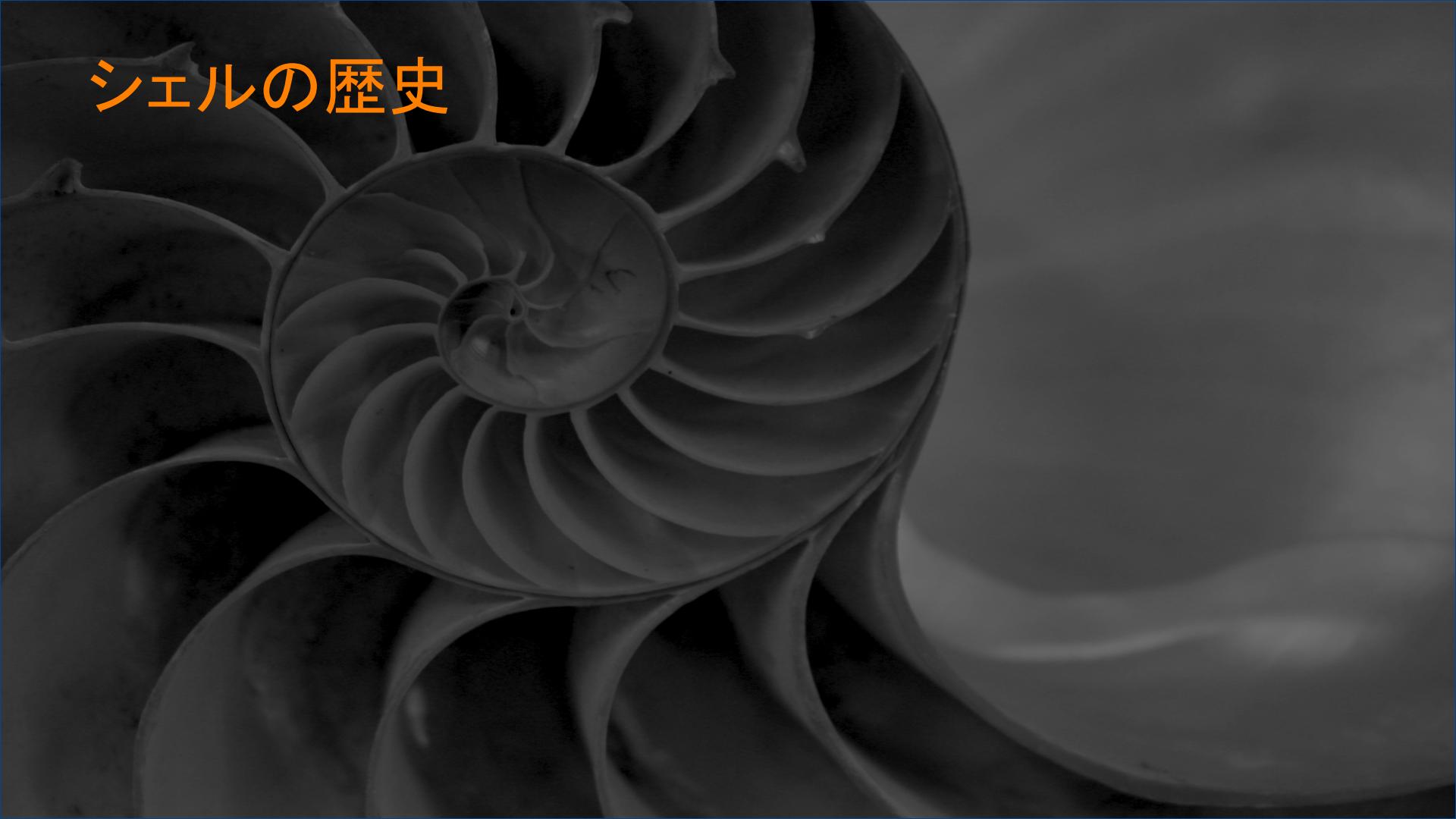
# シェルとは

- ・ 大きく分けて 2 つある
  - CLI (コマンドラインインターフェースを提供する)
  - GUI (グラフィカルインターフェースを提供する)
- ・ どちらにせよ、共通して「**プログラムの起動**」を担う
- ・ 今回扱うのは、CLI のシェル
  - 中でも UNIX シェルと呼ばれるもの
  - MS-DOS やその他 OS のシェルも存在する
- ・ 余談

# シェルとは

- 大きく分けて 2 つある
  - CLI (コマンドラインインターフェースを提供する)
  - GUI (グラフィカルインターフェースを提供する)
- どちらにせよ、共通して「**プログラムの起動**」を担う
- 今回扱うのは、CLI のシェル
  - 中でも UNIX シェルと呼ばれるもの
  - MS-DOS やその他 OS のシェルも存在する
- 余談
  - 「シェル」という名称は POSIX 系 OS のもの
  - 一般には「**コマンドラインインタプリタ**」である

# シェルの歴史



# シェルの歴史

- sh (Tompson shell)
  - 最初に生まれたシェル
  - 最初の UNIX に搭載された

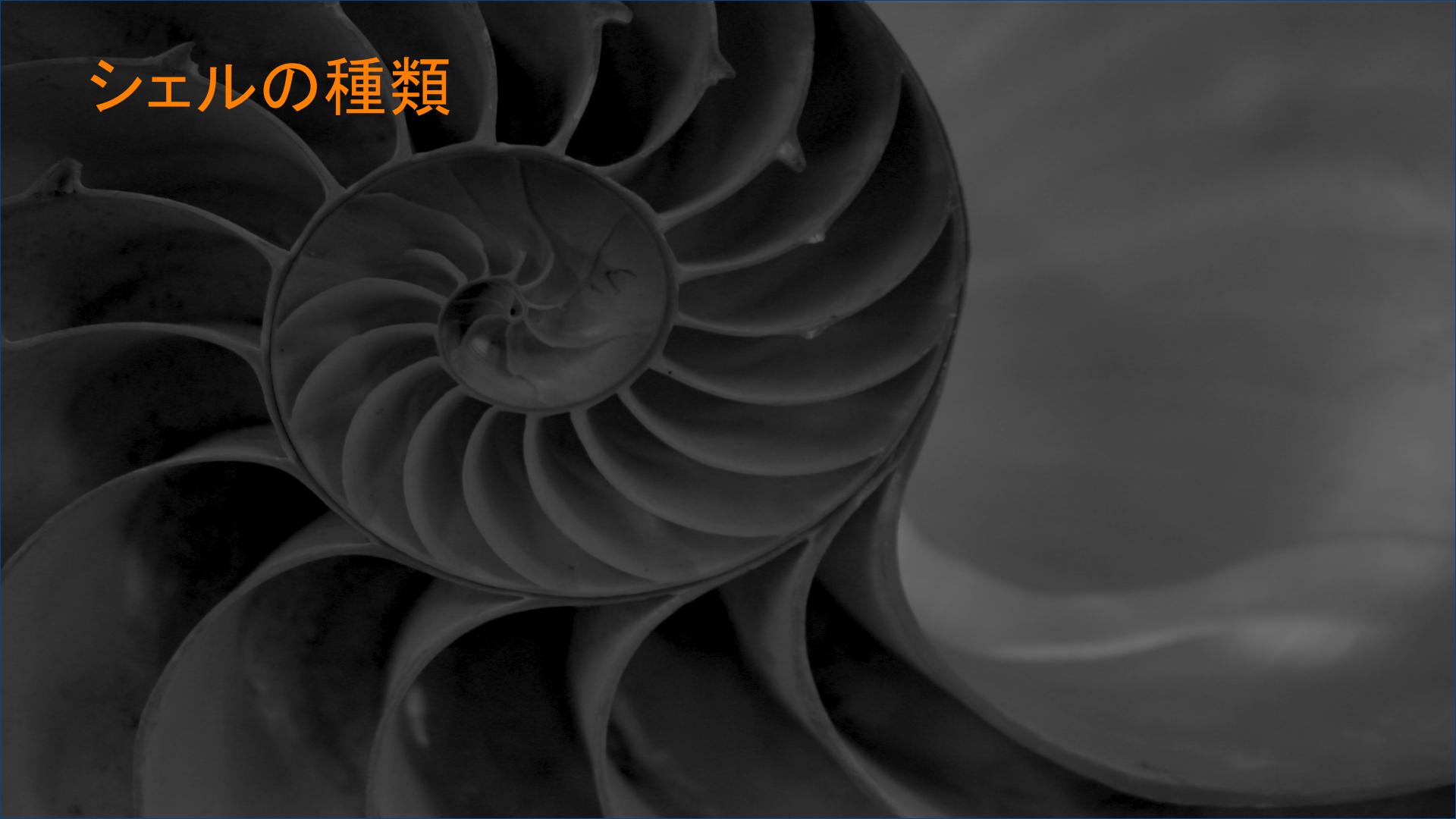
# シェルの歴史

- sh (Tompson shell)
  - 最初に生まれたシェル
  - 最初の UNIX に搭載された
- Bourne Shell
  - B Shell や単に sh と称される
  - 多くのシェルに影響を与えた
  - パイプ, コマンド置換, 条件分岐, ループ, ワイルドカード, ヒアドキュメント

# シェルの歴史

- sh (Tompson shell)
  - 最初に生まれたシェル
  - 最初の UNIX に搭載された
- Bourne Shell
  - B Shell や単に sh と称される
  - 多くのシェルに影響を与えた
  - パイプ, コマンド置換, 条件分岐, ループ, ワイルドカード, ヒアドキュメント
- C Shell
  - ビル・ジョイによって開発された C 言語をモデルとした制御構造を持つ
  - ヒストリ, 編集機構, エイリアス, ディレクトリスタック, cdpth, ジョブ制御

# シェルの種類



# シェルの種類

- Bourne Shell 互換
- C Shell 互換

# シェルの種類

- Bourne Shell 互換

# シェルの種類

- Bourne Shell 互換
  - Bourne Shell(sh)
    - 1978年ごろ V7 Unix の一部として配布され, 大きく普及する

# シェルの種類

- Bourne Shell 互換
  - Bourne Shell(sh)
    - 1978年ごろ V7 Unix の一部として配布され、大きく普及する
  - Bourne-Again shell(bash)
    - Bourne Shell のオープンソース版として GNU プロジェクトが手がけている。多くの Linux Distribution でデフォルトシェルとされている

# シェルの種類

- Bourne Shell 互換
  - Bourne Shell(sh)
    - 1978年ごろ V7 Unix の一部として配布され、大きく普及する
  - Bourne-Again shell(bash)
    - Bourne Shell のオープンソース版として GNU プロジェクトが手がけている。多くの Linux Distribution でデフォルトシェルとされている
  - Almquist Shell(ash)
    - Bourne Shell の BSD ライセンス版として開発。軽量なためリソースが少ない環境で使われる

# シェルの種類

- Bourne Shell 互換
  - Bourne Shell(sh)
    - 1978年ごろ V7 Unix の一部として配布され、大きく普及する
  - Bourne-Again shell(bash)
    - Bourne Shell のオープンソース版として GNU プロジェクトが手がけている。多くの Linux Distribution でデフォルトシェルとされている
  - Almquist Shell(ash)
    - Bourne Shell の BSD ライセンス版として開発。軽量なためリソースが少ない環境で使われる
  - Debian Almquist shell(dash)
    - Debian と Ubuntu で ash の代替としてデフォルトになっている

# シェルの種類

- Bourne Shell 互換
  - Bourne Shell(sh)
    - 1978年ごろ V7 Unix の一部として配布され、大きく普及する
  - Bourne-Again shell(bash)
    - Bourne Shell のオープンソース版として GNU プロジェクトが手がけている。多くの Linux Distribution でデフォルトシェルとされている
  - Almquist Shell(ash)
    - Bourne Shell の BSD ライセンス版として開発。軽量なためリソースが少ない環境で使われる
  - Debian Almquist shell(dash)
    - Debian と Ubuntu で ash の代替としてデフォルトになっている
  - Korn Shell(ksh)
    - 商用 UNIX 系に多く搭載されたシェル

# シェルの種類

- Bourne Shell 互換
  - Bourne Shell(sh)
    - 1978年ごろ V7 Unix の一部として配布され、大きく普及する
  - Bourne-Again shell(bash)
    - Bourne Shell のオープンソース版として GNU プロジェクトが手がけている。多くの Linux Distribution でデフォルトシェルとされている
  - Almquist Shell(ash)
    - Bourne Shell の BSD ライセンス版として開発。軽量なためリソースが少ない環境で使われる
  - Debian Almquist shell(dash)
    - Debian と Ubuntu で ash の代替としてデフォルトになっている
  - Korn Shell(ksh)
    - 商用 UNIX 系に多く搭載されたシェル
  - Z Shell(zsh)
    - 最も高機能なシェルで sh, ash, bash, csh, ksh, tcsh の上位互換機能を持つ

# シェルの種類

- C Shell 互換

# シェルの種類

- C Shell 互換
  - C Shell(csh)
    - ビル・ジョイがカルフォルニア大学バークレイ校で開発した C 言語に似た文法特性を持つシェル. BSD 系に多く搭載された

# シェルの種類

- C Shell 互換
  - C Shell(csh)
    - ビル・ジョイがカルフォルニア大学バークレイ校で開発した C 言語に似た文法特性を持つシェル。BSD 系に多く搭載された
  - TENEX C shell(tcsh)
    - ユーザインターフェースを向上させた csh の上位互換。シェル界隈でいち早く国際化(Native Language System)に対応した。FreeBSD に標準搭載され、その流れを汲む Mac OS X にも標準搭載されていた(/bin/tcsh)。10.3 以降は Linux のデフォルトシェルである Bash がデフォルトになっている

ログインシェル



# ログインシェル

- ユーザがログイン後に使うシェルのこと(デフォルトのシェル)

# ログインシェル

- ユーザがログイン後に使うシェルのこと(デフォルトのシェル)
- 環境変数 \$SHELL に設定されている

# ログインシェル

- ユーザがログイン後に使うシェルのこと(デフォルトのシェル)
- 環境変数 \$SHELL に設定されている

```
$ echo $SHELL  
/bin/bash
```

# ログインシェル

- ユーザがログイン後に使うシェルのこと(デフォルトのシェル)
- 環境変数 `$SHELL` に設定されている

```
$ echo $SHELL  
/bin/bash
```

- Linux や OS X なら bash, FreeBSD なら tcsh

# ログインシェル

- ユーザがログイン後に使うシェルのこと(デフォルトのシェル)
- 環境変数 `$SHELL` に設定されている

```
$ echo $SHELL  
/bin/bash
```

- Linux や OS X なら bash, FreeBSD なら tcsh
- デファクトスタンダードになりつつある `bash` を使うのがおすすめ

# ログインシェル

- ユーザがログイン後に使うシェルのこと(デフォルトのシェル)
- 環境変数 `$SHELL` に設定されている

```
$ echo $SHELL  
/bin/bash
```

- Linux や OS X なら bash, FreeBSD なら tcsh
- デファクトスタンダードになりつつある `bash` を使うのがおすすめ
  - `bash` さえ知っていれば、汎用性高く同じ操作方法で扱える

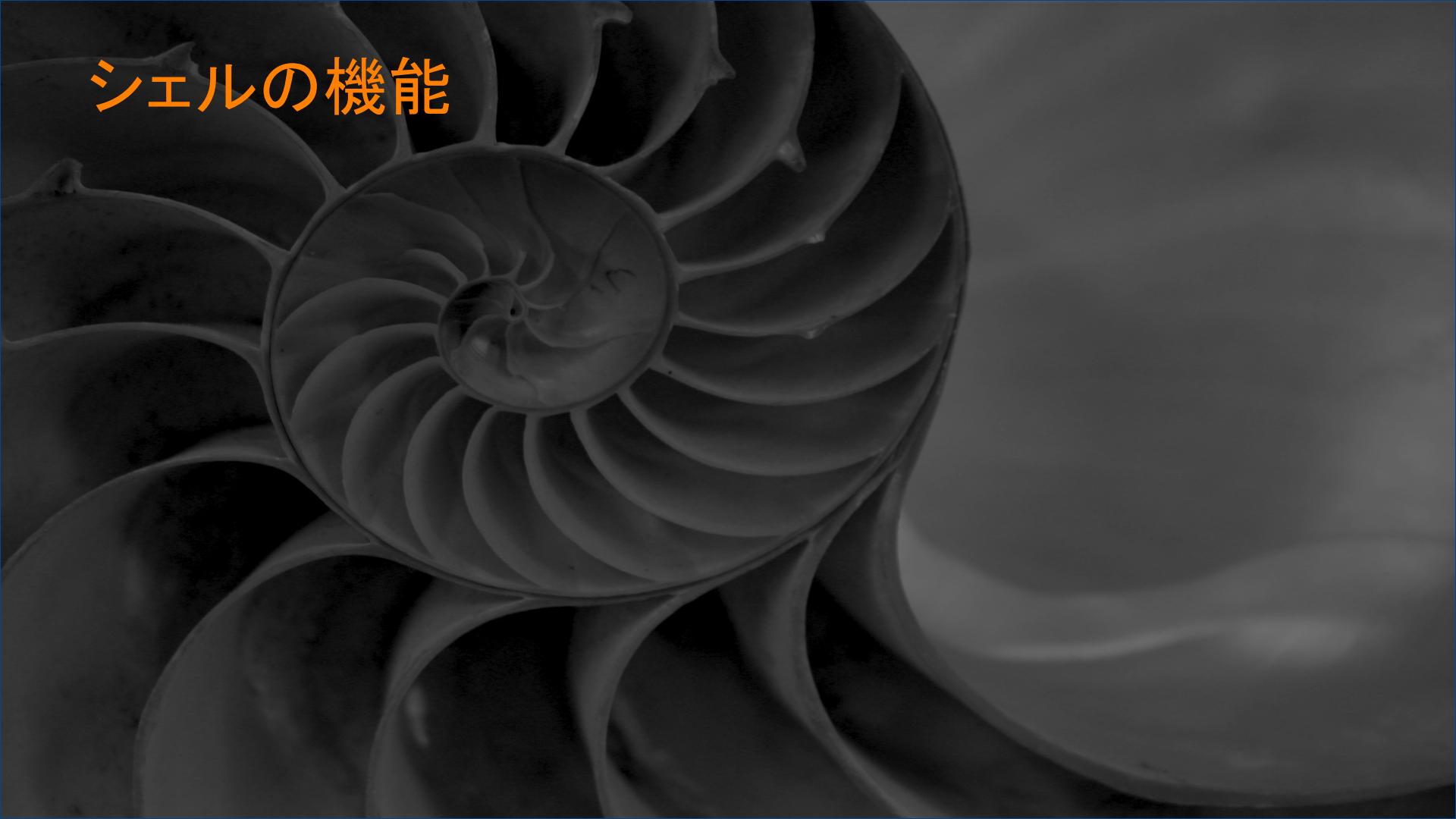
# ログインシェル

- ユーザがログイン後に使うシェルのこと(デフォルトのシェル)
- 環境変数 `$SHELL` に設定されている

```
$ echo $SHELL  
/bin/bash
```

- Linux や OS X なら bash, FreeBSD なら tcsh
- デファクトスタンダードになりつつある `bash` を使うのがおすすめ
  - `bash` さえ知っていれば、汎用性高く同じ操作方法で扱える
- 上級者は更なるユーザビリティを求めて,  
`Z Shell(zsh)` や `Friendly Interactive Shell(fish)` もあり

# シェルの機能



# シェルの機能

- ・ 以下

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定
  - glob によるパターンマッチの展開(ワイルドカード)

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定
  - glob によるパターンマッチの展開(ワイルドカード)
  - ヒストリの呼び出しやその編集(コマンド入力ヒストリ)

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定
  - glob によるパターンマッチの展開(ワイルドカード)
  - ヒストリの呼び出しやその編集(コマンド入力ヒストリ)
  - コマンドに別名を付ける(エイリアス)

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定
  - glob によるパターンマッチの展開(ワイルドカード)
  - ヒストリの呼び出しやその編集(コマンド入力ヒストリ)
  - コマンドに別名を付ける(エイリアス)
  - 繰り返し実行や, 条件分岐実行(制御構造)

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定
  - glob によるパターンマッチの展開(ワイルドカード)
  - ヒストリの呼び出しやその編集(コマンド入力ヒストリ)
  - コマンドに別名を付ける(エイリアス)
  - 繰り返し実行や, 条件分岐実行(制御構造)
  - コマンド入力時のファイルなどの補完機能

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定
  - glob によるパターンマッチの展開(ワイルドカード)
  - ヒストリの呼び出しやその編集(コマンド入力ヒストリ)
  - コマンドに別名を付ける(エイリアス)
  - 繰り返し実行や, 条件分岐実行(制御構造)
  - コマンド入力時のファイルなどの補完機能
  - まとめた一連のコマンドのバッチ処理(シェルスクリプト)

# シェルの機能

- 以下
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定
  - glob によるパターンマッチの展開(ワイルドカード)
  - ヒストリの呼び出しやその編集(コマンド入力ヒストリ)
  - コマンドに別名を付ける(エイリアス)
  - 繰り返し実行や, 条件分岐実行(制御構造)
  - コマンド入力時のファイルなどの補完機能
  - まとめた一連のコマンドのバッチ処理(シェルスクリプト)

# シェルの機能

- 以下(ただしすべてのシェルが持つわけではない)
  - プログラムの実行, アプリケーションの起動
  - プログラムの制御, フォアグラウンド・バックグラウンド(ジョブコントロール)
  - プログラムの出力をファイルへ(リダイレクト)
  - プログラムの出力を他のプログラムの入力にする(パイプ)
  - 環境変数, シェル変数の参照・設定
  - glob によるパターンマッチの展開(ワイルドカード)
  - ヒストリの呼び出しやその編集(コマンド入力ヒストリ)
  - コマンドに別名を付ける(エイリアス)
  - 繰り返し実行や, 条件分岐実行(制御構造)
  - コマンド入力時のファイルなどの補完機能
  - まとめた一連のコマンドのバッチ処理(シェルスクリプト)



シェルスクリプト

# シェルスクリプト

- シェルの大きな特徴

# シェルスクリプト

- シェルの大きな特徴
  - コマンドラインの指示のための変数や制御機能を持つこと
  - プログラミングの類と呼べる複雑性

# シェルスクリプト

- シェルの大きな特徴
  - コマンドラインの指示のための変数や制御機能を持つこと
  - プログラミングの類と呼べる複雑性

```
for file in *.txt
do
    cp "$file" "${file}.bak"
done
```

# シェルスクリプト

- ・ シェルの大きな特徴
  - コマンドラインの指示のための変数や制御機能を持つこと
  - プログラミングの類と呼べる複雑性

```
for file in *.txt
do
    cp "$file" "${file}.bak"
done
```

- これをコマンドラインから入力することもできる

# シェルスクリプト

- ・ シェルの大きな特徴
  - コマンドラインの指示のための変数や制御機能を持つこと
  - プログラミングの類と呼べる複雑性

```
for file in *.txt
do
    cp "$file" "${file}.bak"
done
```

- これをコマンドラインから入力することもできる
- ファイルに保存して実行権限を付けて実行することもできる

# シェルスクリプト

- ・ シェルの大きな特徴
  - コマンドラインの指示のための変数や制御機能を持つこと
  - プログラミングの類と呼べる複雑性

```
for file in *.txt
do
    cp "$file" "${file}.bak"
done
```

- これをコマンドラインから入力することもできる
- ファイルに保存して実行権限を付けて実行することもできる
- シェルスクリプトの基本

# シェルスクリプトの特徴

- バッチ処理に向く

# シェルスクリプトの特徴

- バッチ処理に向く
  - シェルスクリプトを使えば、コマンドラインインターフェースで人手で入力していたコマンド列を自動的に実行でき、一連のコマンドを連続的に実行できる

# シェルスクリプトの特徴

- バッチ処理に向く
  - シェルスクリプトを使えば、コマンドラインインターフェースで人手で入力していたコマンド列を自動的に実行でき、一連のコマンドを連続的に実行できる
- シバン(シェバン)について

# シェルスクリプトの特徴

- バッチ処理に向く
  - シェルスクリプトを使えば、コマンドラインインターフェースで人手で入力していたコマンド列を自動的に実行でき、一連のコマンドを連続的に実行できる
- シバン(シェバン)について
  - 省略

# シェルスクリプトの特徴

- バッチ処理に向く
  - シェルスクリプトを使えば、コマンドラインインターフェースで人手で入力していたコマンド列を自動的に実行でき、一連のコマンドを連続的に実行できる
- シバン(シェバン)について
  - 省略
- プログラミング言語としての側面

# シェルスクリプトの特徴

- バッチ処理に向く
  - シェルスクリプトを使えば、コマンドラインインターフェースで人手で入力していたコマンド列を自動的に実行でき、一連のコマンドを連続的に実行できる
- シバン(シェバン)について
  - 省略
- プログラミング言語としての側面
  - 現代のシェルは汎用プログラミング言語としての機能を持つ

# シェルスクリプトの特徴

- バッチ処理に向く
  - シェルスクリプトを使えば、コマンドラインインターフェースで人手で入力していたコマンド列を自動的に実行でき、一連のコマンドを連續的に実行できる
- シバン(シェバン)について
  - 省略
- プログラミング言語としての側面
  - 現代のシェルは汎用プログラミング言語としての機能を持つ
  - 制御構造、変数、配列、コメント、関数

# シェルスクリプトの特徴

- ・ バッチ処理に向く
  - シェルスクリプトを使えば、コマンドラインインターフェースで人手で入力していたコマンド列を自動的に実行でき、一連のコマンドを連續的に実行できる
- ・ シバン(シェバン)について
  - 省略
- ・ プログラミング言語としての側面
  - 現代のシェルは汎用プログラミング言語としての機能を持つ
  - 制御構造、変数、配列、コメント、関数 → 高機能なアプリケーションが作れる

# シェルスクリプトの特徴

- ・ バッチ処理に向く
  - シェルスクリプトを使えば、コマンドラインインターフェースで人手で入力していたコマンド列を自動的に実行でき、一連のコマンドを連続的に実行できる
- ・ シバン(シェバン)について
  - 省略
- ・ プログラミング言語としての側面
  - 現代のシェルは汎用プログラミング言語としての機能を持つ
  - 制御構造、変数、配列、コメント、関数 → 高機能なアプリケーションが作れる
  - 一方で高水準言語が持つ型システム、スレッド、クラス、高度な科学計算はない
  - また、性能重視のインタプリタ(Ruby, Python)にも劣る
  - 開発のライフサイクルの初期段階にも用いられる。最初はシェルスクリプトでプロトタイプを作成し、Python や C 言語などで書き換えてくこともある

# シェルスクリプトの特徴

- ・ 長所

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所
  - 気をつけないと手痛いエラーが起こりやすい(型がない)

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所
  - 気をつけないと手痛いエラーが起こりやすい(型がない)
  - 変数のチェックなどしっかり行わないといけない

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所
  - 気をつけないと手痛いエラーが起こりやすい(型がない)
  - 変数のチェックなどしっかり行わないといけない
  - cp や mv, rm などでうっかりファイルを消してしまうことがある

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所
  - 気をつけないと手痛いエラーが起こりやすい(型がない)
  - 変数のチェックなどしっかり行わないといけない
  - cp や mv, rm などでうっかりファイルを消してしまうことがある
  - プロセスを意識して書かないと実行速度が遅くなる

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所
  - 気をつけないと手痛いエラーが起こりやすい(型がない)
  - 変数のチェックなどしっかり行わないといけない
  - cp や mv, rm などでうっかりファイルを消してしまうことがある
  - プロセスを意識して書かないと実行速度が遅くなる
  - ほぼすべてのシェルコマンドの実行はプロセスを新たに作り出す(パイプも)

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所
  - 気をつけないと手痛いエラーが起こりやすい(型がない)
  - 変数のチェックなどしっかり行わないといけない
  - cp や mv, rm などでうっかりファイルを消してしまうことがある
  - プロセスを意識して書かないと実行速度が遅くなる
  - ほぼすべてのシェルコマンドの実行はプロセスを新たに作り出す(パイプも)
  - シェル間のプラットフォームの問題

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所
  - 気をつけないと手痛いエラーが起こりやすい(型がない)
  - 変数のチェックなどしっかり行わないといけない
  - cp や mv, rm などでうっかりファイルを消してしまうことがある
  - プロセスを意識して書かないと実行速度が遅くなる
  - ほぼすべてのシェルコマンドの実行はプロセスを新たに作り出す(パイプも)
  - シェル間のプラットフォームの問題
  - 小さな方言や POSIX など

# シェルスクリプトの特徴

- 長所
  - 同じプログラムを書く場合、他の言語より早く短く書けることが多い
  - ファイル操作機能も豊富で、素早く実行でき、対話的デバッグも簡単
  - コンパイルが不要でインタプリタ実行による
- 短所
  - 気をつけないと手痛いエラーが起こりやすい(型がない)
  - 変数のチェックなどしっかり行わないといけない
  - cp や mv, rm などでうっかりファイルを消してしまうことがある
  - プロセスを意識して書かないと実行速度が遅くなる
  - ほぼすべてのシェルコマンドの実行はプロセスを新たに作り出す(パイプも)
  - シェル間のプラットフォームの問題
  - 小さな方言や POSIX など

# シェルスクリプトの種類



# シェルスクリプトの種類

- ・ シェルの数だけシェルスクリプトを書くことができる

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから
- ただし, bash で書くことを推奨する

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから
- ただし, bash で書くことを推奨する
  - 移植性を考慮したら, POSIX sh が書くのがよいが説明が必要になるので今回は省略して bash に統一する

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから
- ただし, bash で書くことを推奨する
  - 移植性を考慮したら, POSIX sh が書くのがよいが説明が必要になるので今回は省略して bash に統一する
  - bash はシェルの中でも枯れた技術であるし, GNU/Linux のデフォルトシェルである

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから
- ただし, bash で書くことを推奨する
  - 移植性を考慮したら, POSIX sh が書くのがよいが説明が必要になるので今回は省略して bash に統一する
  - bash はシェルの中でも枯れた技術であるし, GNU/Linux のデフォルトシェルである
- C Shell 系で書くことはあり得ない

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから
- ただし, bash で書くことを推奨する
  - 移植性を考慮したら, POSIX sh が書くのがよいが説明が必要になるので今回は省略して bash に統一する
  - bash はシェルの中でも枯れた技術であるし, GNU/Linux のデフォルトシェルである
- C Shell 系で書くことはあり得ない
  - ログインシェルには向くがシェルスクリプトとしてはクオリティが低い

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから
- ただし, bash で書くことを推奨する
  - 移植性を考慮したら, POSIX sh が書くのがよいが説明が必要になるので今回は省略して bash に統一する
  - bash はシェルの中でも枯れた技術であるし, GNU/Linux のデフォルトシェルである
- C Shell 系で書くことはあり得ない
  - ログインシェルには向くがシェルスクリプトとしてはクオリティが低い
- Z Shell はあり得なくはないが移植性は二の次

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから
- ただし, bash で書くことを推奨する
  - 移植性を考慮したら, POSIX sh が書くのがよいが説明が必要になるので今回は省略して bash に統一する
  - bash はシェルの中でも枯れた技術であるし, GNU/Linux のデフォルトシェルである
- C Shell 系で書くことはあり得ない
  - ログインシェルには向くがシェルスクリプトとしてはクオリティが低い
- Z Shell はあり得なくはないが移植性は二の次
  - bash よりも高機能な操作や文字列処理が可能

# シェルスクリプトの種類

- シェルの数だけシェルスクリプトを書くことができる
  - シェルはインタプリタだから
- ただし, bash で書くことを推奨する
  - 移植性を考慮したら, POSIX sh が書くのがよいが説明が必要になるので今回は省略して bash に統一する
  - bash はシェルの中でも枯れた技術であるし, GNU/Linux のデフォルトシェルである
- C Shell 系で書くことはあり得ない
  - ログインシェルには向くがシェルスクリプトとしてはクオリティが低い
- Z Shell はあり得なくはないが移植性は二の次
  - bash よりも高機能な操作や文字列処理が可能
  - ただし、まだまだインストールされていない環境も多い

# シェルスクリプトの文法



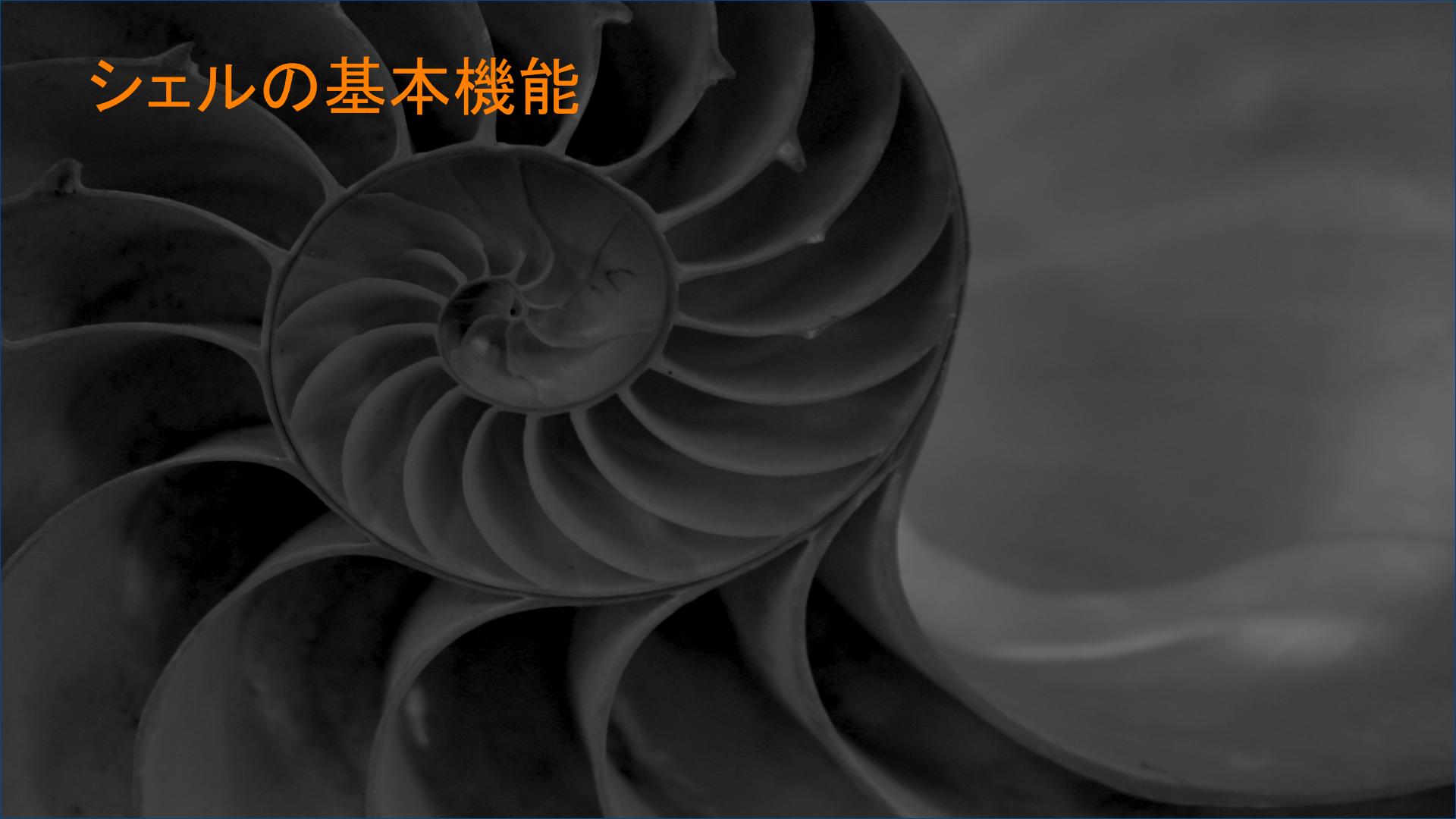
# シェルスクリプトの文法

- シェルの基本機能
- 制御文と test コマンド
- シェルの組み込みコマンド
- 変数と関数

# シェルスクリプトの文法

- シェルの基本機能
- 制御文と test コマンド
- シェルの組み込みコマンド
- 変数と関数

# シェルの基本機能



# シェルの基本機能

- 変数

# 变数

- 变数名

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)
- 変数の宣言

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)
- 変数の宣言
  - 特に宣言なく作れる(意図的に宣言することもできる)

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)
- 変数の宣言
  - 特に宣言なく作れる(意図的に宣言することもできる)
  - `var=abcd`

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)
- 変数の宣言
  - 特に宣言なく作れる(意図的に宣言することもできる)
  - `var=abcd`
  - イコールの間にスペースは入れられない

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)
- 変数の宣言
  - 特に宣言なく作れる(意図的に宣言することもできる)
  - `var=abcd`
  - イコールの間にスペースは入れられない
  - クオートやダブルクオートで囲うこともできる

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)
- 変数の宣言
  - 特に宣言なく作れる(意図的に宣言することもできる)
  - `var=abcd`
  - イコールの間にスペースは入れられない
  - クオートやダブルクオートで囲うこともできる
  - `var="abcd"`

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)
- 変数の宣言
  - 特に宣言なく作れる(意図的に宣言することもできる)
  - `var=abcd`
  - イコールの間にスペースは入れられない
  - クオートやダブルクオートで囲うこともできる
  - `var="abcd"`
  - この場合はなくても構わないが空白やメタキャラクタを含む場合などクオーテーションが必要な場合もある

# 変数

- 変数名
  - 変数に使用できる文字は英数字とアンダースコア(数字から始まる変数は作れない)
- 変数の宣言
  - 特に宣言なく作れる(意図的に宣言することもできる)
  - `var=abcd`
  - イコールの間にスペースは入れられない
  - クオートやダブルクオートで囲うこともできる
  - `var="abcd"`
  - この場合はなくても構わないが空白やメタキャラクタを含む場合などクオーテーションが必要な場合もある
  - 変数の宣言は値を代入したときに行うのが一般的である

# 変数

- ・ 変数の値を参照する

# 変数

- 変数の値を参照する

- \$変数名
- \${変数名}

# 変数

- 変数の値を参照する
  - \${変数名}
  - \${{変数名}}
  - どこまでが変数かを明示する場合は後者である必要がある

# 変数

- 変数の値を参照する
  - `$変数名`
  - `${変数名}`
  - どこまでが変数かを明示する場合は後者である必要がある
  - 中身を確認するのは `echo` コマンドを用いる

# 変数

- 変数の値を参照する
  - \$変数名
  - \${変数名}
  - どこまでが変数かを明示する場合は後者である必要がある
  - 中身を確認するのは echo コマンドを用いる
  - echo \$hoge

# 変数

- 変数の値を参照する
  - `$変数名`
  - `${変数名}`
  - どこまでが変数かを明示する場合は後者である必要がある
  - 中身を確認するのは `echo` コマンドを用いる
  - `echo $hoge`
  - ブレースが必要になるには次の例

# 変数

- 変数の値を参照する
  - `$変数名`
  - `${変数名}`
  - どこまでが変数かを明示する場合は後者である必要がある
  - 中身を確認するのは `echo` コマンドを用いる
  - `echo $hoge`
  - ブレースが必要になるには次の例
  - `echo $hogefuga`
  - `echo ${hoge}fuga`

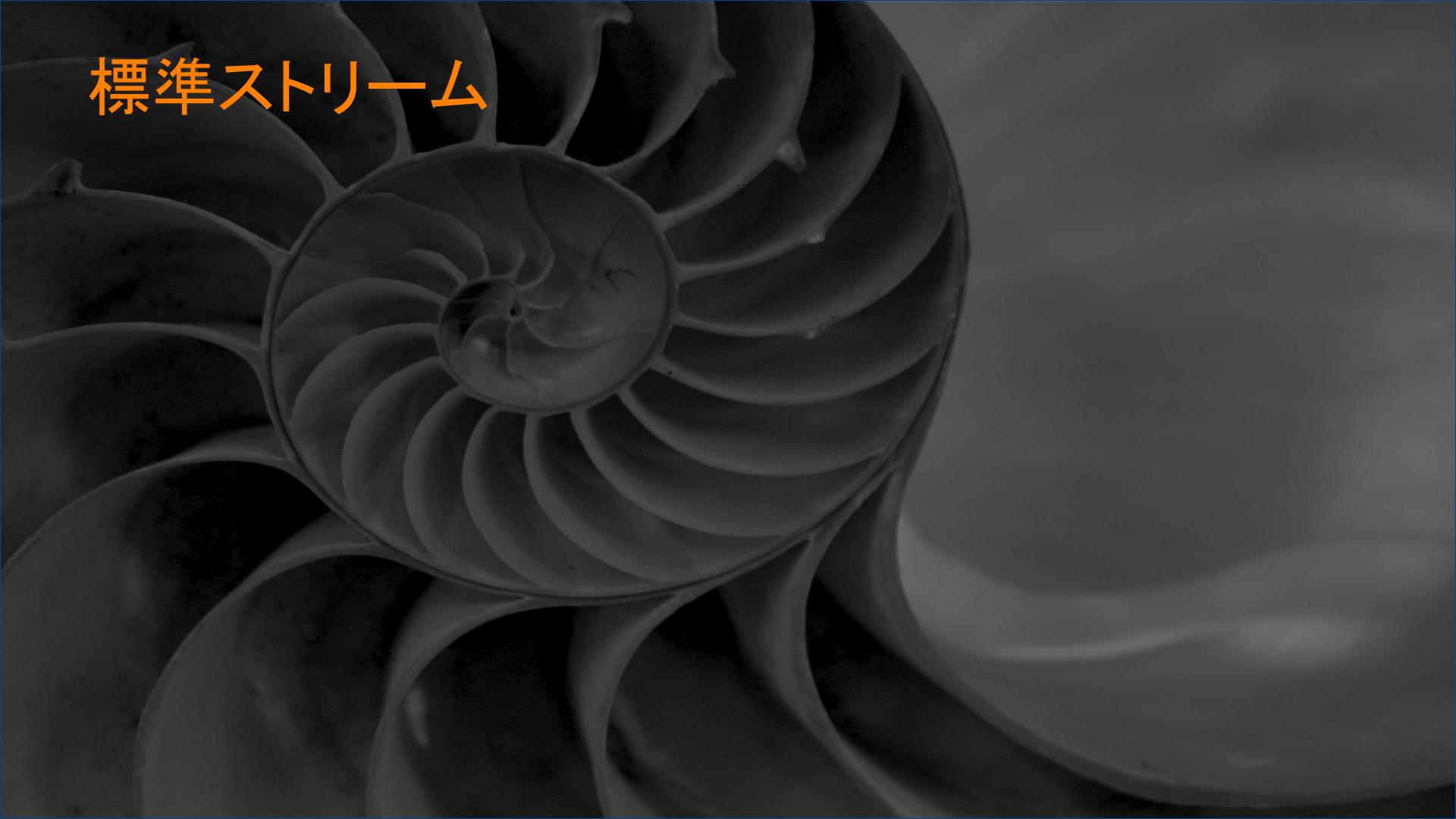
# 変数

- 変数の値を参照する
  - `$変数名`
  - `${変数名}`
  - どこまでが変数かを明示する場合は後者である必要がある
  - 中身を確認するのは `echo` コマンドを用いる
  - `echo $hoge`
  - ブレースが必要になるには次の例
  - `echo $hogefuga` → ✗
  - `echo ${hoge}fuga` → ◎

# シェルの基本機能

- 変数
- 標準ストリーム

標準ストリーム



# 標準ストリーム

- Unix 系 OS や一部プログラミング言語インターフェースでプログラムと端末に接続している入出力チャネルのこと

# 標準ストリーム

- Unix 系 OS や一部プログラミング言語インターフェースでプログラムと端末に接続している入出力チャネルのこと
- つまり、ユーザとシェルなどがやり取りをする際の普遍的な共通インターフェースのこと

# 標準ストリーム

- Unix 系 OS や一部プログラミング言語インターフェースでプログラムと端末に接続している入出力チャネルのこと
- つまり、ユーザとシェルなどがやり取りをする際の普遍的な共通インターフェースのこと
- 現在では 3 つのチャネルがある

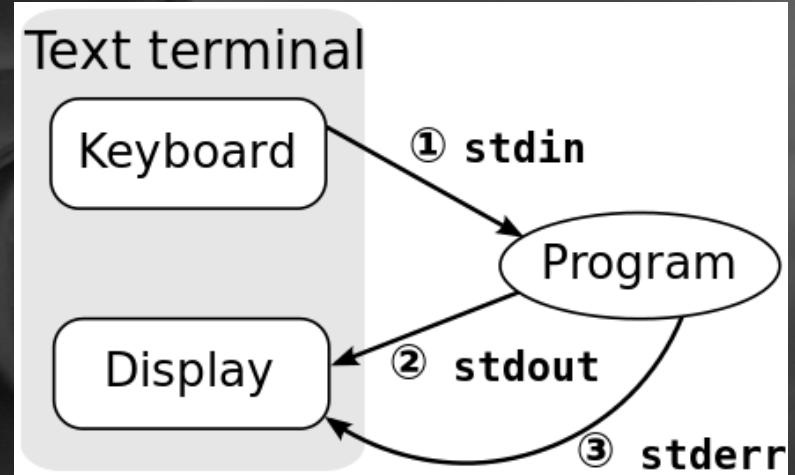
# 標準ストリーム

- Unix 系 OS や一部プログラミング言語インターフェースでプログラムと端末に接続している入出力チャネルのこと
- つまり、ユーザとシェルなどがやり取りをする際の普遍的な共通インターフェースのこと
- 現在では 3 つのチャネルがある
  - 標準入力(`stdin`)
  - 標準出力(`stdout`)
  - 標準エラー出力(`stderr`)

# 標準ストリーム

- Unix 系 OS や一部プログラミング言語インターフェースでプログラムと端末に接続している入出力チャネルのこと
- つまり、ユーザとシェルなどがやり取りをする際の普遍的な共通インターフェースのこと
- 現在では 3 つのチャネルがある
  - 標準入力(`stdin`)
  - 標準出力(`stdout`)
  - 標準エラー出力(`stderr`)

標準ストリーム - Wikipedia



# 標準ストリーム

- 標準入力(stdin)

# 標準ストリーム

- 標準入力 (stdin)
  - シェルに対するキーボードからの入力

# 標準ストリーム

- 標準入力(stdin)
  - シェルに対するキーボードからの入力
  - プログラムに入ってくるデータ(テキストであることが多い)

# 標準ストリーム

- 標準入力(stdin)
  - シェルに対するキーボードからの入力
  - プログラムに入ってくるデータ(テキストであることが多い)
  - すべてのプログラムが入力を要求するわけではない
    - 例えば ls は標準入力を必要とせず実行が完了する

# 標準ストリーム

- 標準入力 (stdin)
  - シェルに対するキーボードからの入力
  - プログラムに入ってくるデータ(テキストであることが多い)
  - すべてのプログラムが入力を要求するわけではない
    - 例えば `ls` は標準入力を必要とせず実行が完了する
- 標準出力 (stdout)

# 標準ストリーム

- 標準入力(stdin)
  - シェルに対するキーボードからの入力
  - プログラムに入ってくるデータ(テキストであることが多い)
  - すべてのプログラムが入力を要求するわけではない
    - 例えば ls は標準入力を必要とせず実行が完了する
- 標準出力(stdout)
  - プログラムが(端末画面に)書き出すデータのストリーム

# 標準ストリーム

- 標準入力(stdin)
  - シェルに対するキーボードからの入力
  - プログラムに入ってくるデータ(テキストであることが多い)
  - すべてのプログラムが入力を要求するわけではない
    - 例えば ls は標準入力を必要とせず実行が完了する
- 標準出力(stdout)
  - プログラムが(端末画面に)書き出すデータのストリーム
  - すべてのプログラムが出力を要求するわけではない
    - 例えば mv は何も出力をしない

# 標準ストリーム

- 標準エラー出力(stderr)

# 標準ストリーム

- 標準エラー出力(stderr)
  - エラーレポートを出力するためのストリーム

# 標準ストリーム

- 標準エラー出力(stderr)
  - エラーレポートを出力するためのストリーム
  - デフォルトでは端末画面になっている

# 標準ストリーム

- 標準エラー出力(stderr)
  - エラーレポートを出力するためのストリーム
  - デフォルトでは端末画面になっている
  - 標準出力と出力先が同じなので、一見見間違えるが独立したチャネル

# 標準ストリーム

- 標準エラー出力(stderr)
  - エラーレポートを出力するためのストリーム
  - デフォルトでは端末画面になっている
  - 標準出力と出力先が同じなので、一見見間違えるが独立したチャネル
  - どちらか一方だけを、リダイレクトすることも可能

# 標準ストリーム

- ファイルディスクリプタ(ファイル記述子)

# 標準ストリーム

- ファイルディスクリプタ(ファイル記述子)
  - 標準ストリームを表すシンボルのこと
  - POSIX によると、整数値である

# 標準ストリーム

- ファイルディスクリプタ(ファイル記述子)
  - 標準ストリームを表すシンボルのこと
  - POSIX によると、整数値である

整数値	名前
0	標準入力(stdin)
1	標準出力(stdout)
2	標準エラー出力(stderr)

# 標準ストリーム

- ファイルディスクリプタ(ファイル記述子)
  - 標準ストリームを表すシンボルのこと
  - POSIX によると、整数値である

整数値	名前
0	標準入力(stdin)
1	標準出力(stdout)
2	標準エラー出力(stderr)

- 3番以上はユーザが任意で使用できるファイルディスクリプタ

# 標準ストリーム

- ファイルディスクリプタ(ファイル記述子)
  - 標準ストリームを表すシンボルのこと
  - POSIX によると、整数値である

整数値	名前
0	標準入力(stdin)
1	標準出力(stdout)
2	標準エラー出力(stderr)

- 3番以上はユーザが任意で使用できるファイルディスクリプタ
- 0 - 2はログイン時にシェルによって自動で割り振られる(「標準」入出力)

# シェルの基本機能

- 変数
- 標準ストリーム
- リダイレクション

リダイレクション



# リダイレクション

- ・ リダイレクト

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(<0>), 標準出力(1>), 標準エラー出力(2>)

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0>`), 標準出力(`1>`), 標準エラー出力(`2>`)

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0>`), 標準出力(`1>`), 標準エラー出力(`2>`)
  - ファイルディスクリプタ

# リダイレクション

- リダイレクト

- 標準ストリームをユーザが指定した位置に変更する機能
- 標準入力(`<0>`), 標準出力(`1>`), 標準エラー出力(`2>`)
- ファイルディスクリプタ
- 0と1は省略できる

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0>`), 標準出力(`1>`), 標準エラー出力(`2>`)
  - ファイルディスクリプタ
  - 0と1は省略できる
- null デバイス

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0`), 標準出力(`1>`), 標準エラー出力(`2>`)
  - ファイルディスクリプタ
  - 0と1は省略できる
- null デバイス
  - `/dev/null` のことを指す Unix のスペシャルファイル

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0`), 標準出力(`1>`), 標準エラー出力(`2>`)
  - ファイルディスクリプタ
  - 0と1は省略できる
- null デバイス
  - `/dev/null` のことを指す Unix のスペシャルファイル
  - そこに書き込まれたデータは全て捨て, 読み出してもどんなプロセスに対してもデータを返さない(EOFのみ)

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0`), 標準出力(`1>`), 標準エラー出力(`2>`)
  - ファイルディスクリプタ
  - 0と1は省略できる
- null デバイス
  - `/dev/null` のことを指す Unix のスペシャルファイル
  - そこに書き込まれたデータは全て捨て, 読み出してもどんなプロセスに対してもデータを返さない(EOFのみ)
  - 通称, ブラックホール, ビットバケツ

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0`), 標準出力(`1>`), 標準エラー出力(`2>`)
  - ファイルディスクリプタ
  - 0と1は省略できる
- null デバイス
  - `/dev/null` のことを指す Unix のスペシャルファイル
  - そこに書き込まれたデータは全て捨て, 読み出してもどんなプロセスに対してもデータを返さない(EOFのみ)
  - 通称, ブラックホール, ビットバケツ
  - よく使われるのは, 標準エラー出力だけを捨てたい場合など

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0`), 標準出力(`1>`), 標準エラー出力(`2>`)
  - ファイルディスクリプタ
  - 0と1は省略できる
- null デバイス
  - `/dev/null` のことを指す Unix のスペシャルファイル
  - そこに書き込まれたデータは全て捨て, 読み出してもどんなプロセスに対してもデータを返さない(EOFのみ)
  - 通称, ブラックホール, ビットバケツ
  - よく使われるのは, 標準エラー出力だけを捨てたい場合など
  - `cat file 2>/dev/null`

# リダイレクション

- リダイレクト
  - 標準ストリームをユーザが指定した位置に変更する機能
  - 標準入力(`<0`), 標準出力(`1>`), 標準エラー出力(`2>`)
  - ファイルディスクリプタ
  - 0と1は省略できる
- null デバイス
  - `/dev/null` のことを指す Unix のスペシャルファイル
  - そこに書き込まれたデータは全て捨て, 読み出してもどんなプロセスに対してもデータを返さない(EOFのみ)
  - 通称, ブラックホール, ビットバケツ
  - よく使われるのは, 標準エラー出力だけを捨てたい場合など
  - `cat file 2>/dev/null`
  - これは `file` が存在しない場合のエラー「no such file or directory」を捨てている

# リダイレクション

- 例(標準入出力のリダイレクション)

# リダイレクション

- 例(標準入出力のリダイレクション)
  - `>file` 標準出力の内容をファイル file に書き込む
  - `>>file` 標準出力の内容をファイル file に追記する

# リダイレクション

- 例(標準入出力のリダイレクション)
  - `>file` 標準出力の内容をファイル file に書き込む
  - `>>file` 標準出力の内容をファイル file に追記する
  - `2>file` 標準エラー出力の内容をファイル file に書き込む
  - `2>>file` 標準エラー出力の内容をファイル file に追記する

# リダイレクション

- 例(標準入出力のリダイレクション)
  - `>file` 標準出力の内容をファイル file に書き込む
  - `>>file` 標準出力の内容をファイル file に追記する
  - `2>file` 標準エラー出力の内容をファイル file に書き込む
  - `2>>file` 標準エラー出力の内容をファイル file に追記する
  - `2>&1` 標準エラー出力を標準出力に向ける
  - `1>&2` 標準出力を標準エラー出力に向ける

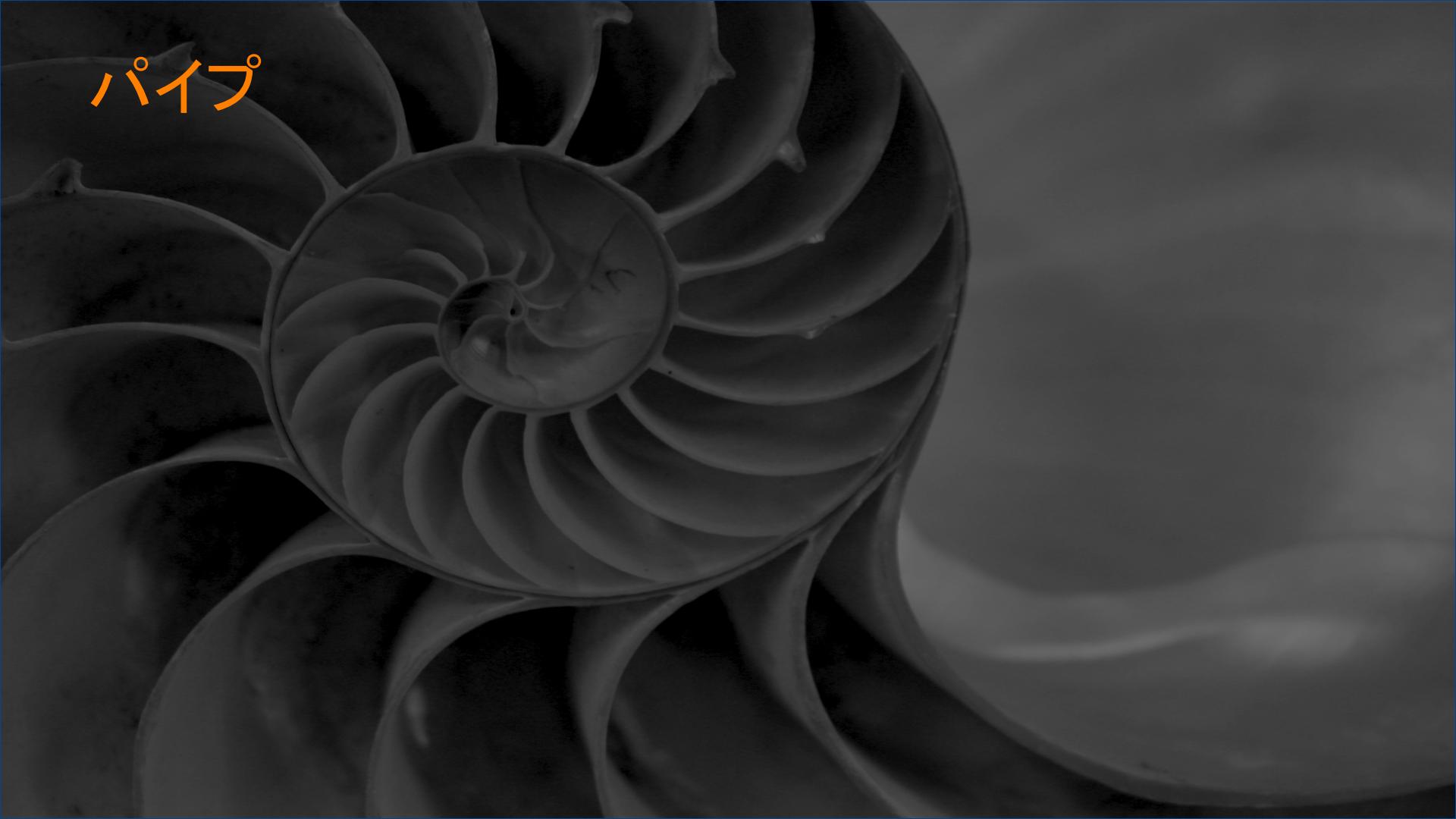
# リダイレクション

- 例(標準入出力のリダイレクション)
  - `>file` 標準出力の内容をファイル file に書き込む
  - `>>file` 標準出力の内容をファイル file に追記する
  - `2>file` 標準エラー出力の内容をファイル file に書き込む
  - `2>>file` 標準エラー出力の内容をファイル file に追記する
  - `2>&1` 標準エラー出力を標準出力に向ける
  - `1>&2` 標準出力を標準エラー出力に向ける
  - `>&m` 標準出力をファイルディスクリプタ m 番に向ける
  - `>&-` 標準出力を閉じる
  - `<file` ファイル file の内容を標準入力にする
  - `<&m` ファイルディスクリプタ m 番が指す対象を標準入力にする
  - `<&-` 標準入力を閉じる

# シェルの基本機能

- 変数
- 標準ストリーム
- リダイレクション
- パイプ

パイプ



# パイプ

- 複数プロセス間の入出力をつなぐ仕組み

# パイプ

- 複数プロセス間の入出力をつなぐ仕組み
- 例

```
- ls -l | grep "pipe" | wc
```

# パイプ

- 複数プロセス間の入出力をつなぐ仕組み
- 例
  - `ls -l | grep "pipe" | wc`
  - これは、(1)ファイルのリストを表示して、(2)「pipe」の名前を持つものに絞り、  
(3)それが何個あるか数えている

# パイプ

- 複数プロセス間の入出力をつなぐ仕組み
- 例
  - `ls -l | grep "pipe" | wc`
  - これは、(1)ファイルのリストを表示して、(2)「pipe」の名前を持つものに絞り、  
(3)それが何個あるか数えている
  - 一つずつファイルに標準出力を書き込み、次のコマンドでそれを標準入力にして  
実行すれば同じことできる

# パイプ

- 複数プロセス間の入出力をつなぐ仕組み
- 例
  - `ls -l | grep "pipe" | wc`
  - これは、(1)ファイルのリストを表示して、(2)「pipe」の名前を持つものに絞り、  
(3)それが何個あるか数えている
  - 一つずつファイルに標準出力を書き込み、次のコマンドでそれを標準入力にして  
実行すれば同じことできる → 大きな手間、中間ファイルが生まれてしまう

# パイプ

- 複数プロセス間の入出力をつなぐ仕組み
- 例
  - `ls -l | grep "pipe" | wc`
  - これは、(1)ファイルのリストを表示して、(2)「pipe」の名前を持つものに絞り、  
(3)それが何個あるか数えている
  - 一つずつファイルに標準出力を書き込み、次のコマンドでそれを標準入力にして  
実行すれば同じことできる → 大きな手間、中間ファイルが生まれてしまう
  - 更に、リダイレクトを用いると処理が遅くなる

# パイプ

- 複数プロセス間の入出力をつなぐ仕組み
- 例
  - `ls -l | grep "pipe" | wc`
  - これは、(1)ファイルのリストを表示して、(2)「pipe」の名前を持つものに絞り、  
(3)それが何個あるか数えている
  - 一つずつファイルに標準出力を書き込み、次のコマンドでそれを標準入力にして  
実行すれば同じことできる → **大きな手間、中間ファイルが生まれてしまう**
  - 更に、リダイレクトを用いると処理が遅くなる
    - これはファイルへの書き出しが完了するまで他の操作が待ち状態に入る

# パイプ

- 複数プロセス間の入出力をつなぐ仕組み
- 例
  - `ls -l | grep "pipe" | wc`
  - これは、(1)ファイルのリストを表示して、(2)「pipe」の名前を持つものに絞り、  
(3)それが何個あるか数えている
  - 一つずつファイルに標準出力を書き込み、次のコマンドでそれを標準入力にして  
実行すれば同じことできる → **大きな手間、中間ファイルが生まれてしまう**
  - 更に、リダイレクトを用いると処理が遅くなる
    - これはファイルへの書き出しが完了するまで他の操作が待ち状態に入る
  - パイプはすべてを並行処理するため、I/O 待ちなどの時間を他に使うことができる

# パイプ

- 複数プロセス間の入出力をつなぐ仕組み
- 例
  - `ls -l | grep "pipe" | wc`
  - これは、(1)ファイルのリストを表示して、(2)「pipe」の名前を持つものに絞り、(3)それが何個あるか数えている
  - 一つずつファイルに標準出力を書き込み、次のコマンドでそれを標準入力にして実行すれば同じことできる → **大きな手間、中間ファイルが生まれてしまう**
  - 更に、リダイレクトを用いると処理が遅くなる
    - これはファイルへの書き出しが完了するまで他の操作が待ち状態に入る
  - パイプはすべてを並行処理するため、I/O 待ちなどの時間を他に使うことができる
  - また、パイプは逐次処理なので、主記憶の利用効率がいい

# パイプ

- エラーストリーム

# パイプ

- エラーストリーム
  - デフォルトでは、標準エラー出力(stderr)はパイプを通して渡されない

# パイプ

- エラーストリーム
  - デフォルトでは、標準エラー出力(stderr)はパイプを通して渡されない
  - パイプを起動した時のエラーストリームに書き出される

# パイプ

- エラーストリーム
  - デフォルトでは、標準エラー出力(stderr)はパイプを通して渡されない
  - パイプを起動した時のエラーストリームに書き出される
  - `cat file | grep "pipe" 2>&1 | less`

# パイプ

- エラーストリーム

- デフォルトでは、標準エラー出力(stderr)はパイプを通して渡されない
- パイプを起動した時のエラーストリームに書き出される
- `cat file | grep "pipe" 2>&1 | less`
- 標準エラー出力(2番)を標準出力(1番)に向けてからパイプに流す

# シェルの基本機能

- 変数
- 標準ストリーム
- リダイレクション
- パイプ
- メタキャラクタ

メタキャラクタ



# メタキャラクタ

- ・ シェルが解釈する特殊文字のこと

# メタキャラクタ

- シェルが解釈する特殊文字のこと

ファイル補完のメタ文字	説明
?	任意の一文字に合致
*	任意の文字列に合致
[str]	文字列 str のどれか 1 文字 に合致. ! で反転, - で範囲
{str, ing}	str, ing いずれかに合致

# メタキャラクタ

- シェルが解釈する特殊文字のこと

ファイル補完のメタ文字	説明
?	任意の一文字に合致
*	任意の文字列に合致
[str]	文字列 str のどれか 1 文字 に合致. ! で反転, - で範囲
{str, ing}	str, ing いずれかに合致

- 正規表現ではない(シェルが理解するワイルドカードの一種)

# シェルスクリプトの文法

- シェルの基本機能 ✓
- 制御文と test コマンド
- シェルの組み込みコマンド
- 変数と関数

# シェルスクリプトの文法

- シェルの基本機能 ✓
- 制御文と test コマンド
- シェルの組み込みコマンド
- 変数と関数

# 制御文と test コマンド



# 制御文と test コマンド

- 制御文

# 制御文と test コマンド

- 制御文
  - if 文
  - for 文
  - while 文
  - case 文
  - until 文
  - select 文

# 制御文と test コマンド

- 制御文
  - if 文
  - for 文
  - while 文
  - case 文
  - until 文
  - select 文

# 制御文と test コマンド

- if 文

# 制御文と test コマンド

- if 文

```
if condition  
then  
    处理1  
elif condition  
then  
    处理2  
else  
    处理3  
fi
```

# 制御文と test コマンド

- if 文

```
if condition  
then  
    処理1  
elif condition  
then  
    処理2  
else  
    処理3  
fi
```

*condition* が真のとき,  
*then* 以下が実行される

# 制御文と test コマンド

- if 文

```
if condition  
then  
    処理1  
elif condition  
then  
    処理2  
else  
    処理3  
fi
```

*condition* が真のとき,  
*then* 以下が実行される

*then* は省略できないが,  
*elif* と *else* は必須ではない

# 制御文と test コマンド

- if 文
- for 文

# 制御文と test コマンド

- for 文

```
for variable in wordlists  
do  
    处理  
done
```

# 制御文と test コマンド

- for 文

```
for variable in wordlists  
do  
    处理  
done
```

他の言語でいう for-each 的な  
書き方

# 制御文と test コマンド

- for 文

```
for variable in wordlists  
do  
    处理  
done
```

他の言語でいう for-each 的な  
書き方

```
for (( expr1; expr2; expr3 ))  
do  
    处理  
done
```

bash 限定で、C 言語風な  
シンタックスも許可されている

# 制御文と test コマンド

- for 文(例)

# 制御文と test コマンド

- for 文(例)

```
for file in ~/.txt  
do  
    cp "$file" "$file".bak  
done
```

# 制御文と test コマンド

- for 文(例)

```
for file in ~/.txt
do
    cp "$file" "$file".bak
done
```

```
for (( i=0; i<10; i++ ))
do
    echo "$i: hello"
done
```

# 制御文と test コマンド

- if 文
- for 文
- while 文

# 制御文と test コマンド

- while 文

# 制御文と test コマンド

- while 文

```
while condition
```

```
do
```

処理

```
done
```

# 制御文と test コマンド

- while 文

```
while condition  
do  
    処理  
done
```

*conditdion* が真である限り,  
処理を実行する

# 制御文と test コマンド

- while 文

```
while condition  
do  
    処理  
done
```

*conditdion* が真である限り,  
処理を実行する

- 無限ループ

# 制御文と test コマンド

- while 文

```
while condition
do
    処理
done
```

*conditdion* が真である限り、  
処理を実行する

- 無限ループ

```
while true
do
    処理
done
```

*true* に入るのは

- *true*
- *test 1*
- *[ 1 ]*
- *:*

# 制御文と test コマンド

- if 文
- for 文
- while 文
- 真偽値

# 制御文と test コマンド

- 真偽値

# 制御文と test コマンド

- 真偽値
  - シェルスクリプトでは 0 が真, 非 0 が偽

# 制御文と test コマンド

- 真偽値
  - シェルスクリプトでは 0 が真, 非 0 が偽
  - `true` コマンドは真を返すだけ, `false` コマンドは偽を返す

# 制御文と test コマンド

- 真偽値
  - シェルスクリプトでは 0 が真, 非 0 が偽
  - `true` コマンドは真を返すだけ, `false` コマンドは偽を返す
  - `:`(コロン)は何もしないコマンド(`true` とだいたい同じ)

# 制御文と test コマンド

- 真偽値
  - シェルスクリプトでは 0 が真, 非 0 が偽
  - `true` コマンドは真を返すだけ, `false` コマンドは偽を返す
  - `:`(コロン)は何もしないコマンド(`true` とだいたい同じ)
  - `test 1` や `[ 1 ]` は常に真を返す

# 制御文と test コマンド

- 真偽値

- シェルスクリプトでは 0 が真, 非 0 が偽
- `true` コマンドは真を返すだけ, `false` コマンドは偽を返す
- `:`(コロン)は何もしないコマンド(`true` とだいたい同じ)
- `test 1` や `[ 1 ]` は常に真を返す
- ちなみに `[` コマンドは `test` コマンドのエイリアス

# 制御文と test コマンド

- 真偽値
  - シェルスクリプトでは 0 が真, 非 0 が偽
  - `true` コマンドは真を返すだけ, `false` コマンドは偽を返す
  - `:`(コロン)は何もしないコマンド(`true` とだいたい同じ)
  - `test 1` や `[ 1 ]` は常に真を返す
  - ちなみに `[` コマンドは `test` コマンドのエイリアス
  - 対応する `]` はコマンドではなく, `[` の最終引数で, 条件の終わりと判断される
  - `[` はコマンドなので前後に空白が必要

# 制御文と test コマンド

- 真偽値
  - シェルスクリプトでは 0 が真, 非 0 が偽
  - `true` コマンドは真を返すだけ, `false` コマンドは偽を返す
  - `:`(コロン)は何もしないコマンド(`true` とだいたい同じ)
  - `test 1` や `[ 1 ]` は常に真を返す
  - ちなみに `[` コマンドは `test` コマンドのエイリアス
  - 対応する `]` はコマンドではなく, `[` の最終引数で, 条件の終わりと判断される
  - `[` はコマンドなので前後に空白が必要
  - `cat file` を `catfile` と出来ない理屈と同じ

# 制御文と test コマンド

- test コマンド

# 制御文と test コマンド

- test コマンド
  - 条件の判定に使う

# 制御文と test コマンド

- test コマンド
  - 条件の判定に使う
  - 数値を比較する
  - 文字列を比較する
  - ファイル形式やファイル情報を判断する
  - 複雑な条件判定をする

# 制御文と test コマンド

- test コマンド
  - 条件の判定に使う
  - 数値を比較する
  - 文字列を比較する
  - ファイル形式やファイル情報を判断する
  - 複雑な条件判定をする

# 制御文と test コマンド

- test コマンド

数値に関する条件	
<code>arg1 -lt arg2</code>	<code>arg1 &lt; arg2</code> のとき真
<code>arg1 -le arg2</code>	<code>arg1 &lt;= arg2</code> のとき真
<code>arg1 -gt arg2</code>	<code>Arg1 &gt; arg2</code> のとき真
<code>arg1 -ge arg2</code>	<code>Arg1 &gt;= arg2</code> のとき真
<code>arg1 -eq arg2</code>	<code>Arg1 == arg2</code> のとき真
<code>arg1 -ne arg2</code>	<code>Arg1 != arg2</code> のとき真

# 制御文と test コマンド

- test コマンド

文字列に関する条件式	
<code>str1 = str2</code>	<code>str1 == str2</code> のとき真
<code>str1 != str2</code>	<code>str1 != str2</code> のとき真
<code>str</code>	文字列が1文字以上のとき真
<code>-n str</code>	文字列が1文字以上のとき真
<code>-z str</code>	文字列が0文字のとき真

# 制御文と test コマンド

- test コマンド

ファイルに関する条件式	
<code>-d file</code>	<i>file</i> がディレクトリ
<code>-f file</code>	<i>file</i> が存在し, 通常ファイル
<code>-h file</code>	<i>file</i> が存在し, シンボリックリンク
<code>-p file</code>	<i>file</i> が存在し, パイプ
<code>-r file</code>	<i>file</i> が存在し, 読み取り可
<code>-w file</code>	<i>file</i> が存在し, 書き込み可
<code>-x file</code>	<i>file</i> が存在し, 実行可能

# 制御文と test コマンド

- test コマンド

条件を論理演算する条件式	
$expr1 -a expr2$	$expr1$ と $expr2$ がどちらも真なら真
$expr1 -o expr2$	$expr1$ か $expr2$ のどちらかが真なら真
$!expr1$	$expr1$ の否定(真なら偽, 偽なら真)
$(expr1)$	()の中の条件式を優先する

# 制御文と test コマンド

- if 文
- for 文
- while 文
- 真偽値
- 短絡評価

# 制御文と test コマンド

- 短絡評価

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる
  - 他の言語でいう三項演算子のようなもの(イコールではない)

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる
  - 他の言語でいう三項演算子のようなもの(イコールではない)
- 例

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる
  - 他の言語でいう三項演算子のようなもの(イコールではない)
- 例
  - `echo "hoge" && echo "fuga"`

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる
  - 他の言語でいう三項演算子のようなもの(イコールではない)
- 例
  - `echo "hoge" && echo "fuga"`
  - && が真の短絡評価

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる
  - 他の言語でいう三項演算子のようなもの(イコールではない)
- 例
  - `echo "hoge" && echo "fuga"`
  - `&&` が真の短絡評価
  - `&&` で挟んだ左項が真なら右項を実行する(偽なら実行されず)

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる
  - 他の言語でいう三項演算子のようなもの(イコールではない)
- 例
  - `echo "hoge" && echo "fuga"`
  - `&&` が真の短絡評価
  - `&&` で挟んだ左項が真なら右項を実行する(偽なら実行されず)
  - 逆は `||`(偽の短絡評価)

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる
  - 他の言語でいう三項演算子のようなもの(イコールではない)
- 例
  - `echo "hoge" && echo "fuga"`
  - `&&` が真の短絡評価
  - `&&` で挟んだ左項が真なら右項を実行する(偽なら実行されず)
  - 逆は `||`(偽の短絡評価)
  - 真なら実行されない

# 制御文と test コマンド

- 短絡評価
  - if 文の書き換えができる
  - 他の言語でいう三項演算子のようなもの(イコールではない)
- 例
  - `echo "hoge" && echo "fuga"`
  - `&&` が真の短絡評価
  - `&&` で挟んだ左項が真なら右項を実行する(偽なら実行されず)
  - 逆は `||`(偽の短絡評価)
  - 真なら実行されない
  - よく使うよ！

# 制御文と test コマンド

- if 文
- for 文
- while 文
- 真偽値
- 短絡評価
- 終了ステータス

# 制御文と test コマンド

- 終了ステータス

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる
  - 全てのコマンドは基本的に終了ステータスを返す

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる
  - 全てのコマンドは基本的に終了ステータスを返す
  - 他の言語で言う戻り値に相当する

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる
  - 全てのコマンドは基本的に終了ステータスを返す
  - 他の言語で言う戻り値に相当する
  - 終了ステータスが 0 か非 0 か, つまり真か偽かを判定して処理を進めていく

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる
  - 全てのコマンドは基本的に終了ステータスを返す
  - 他の言語で言う戻り値に相当する
  - 終了ステータスが 0 か非 0 か, つまり真か偽かを判定して処理を進めていく
  - 先の例の短絡評価もその例の一つ

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる
  - 全てのコマンドは基本的に終了ステータスを返す
  - 他の言語で言う戻り値に相当する
  - 終了ステータスが 0 か非 0 か, つまり真か偽かを判定して処理を進めていく
  - 先の例の短絡評価もその例の一つ
  - もっと言えば, `true && echo` は常に成功し, `true || echo` は常に失敗する

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる
  - 全てのコマンドは基本的に終了ステータスを返す
  - 他の言語で言う戻り値に相当する
  - 終了ステータスが 0 か非 0 か, つまり真か偽かを判定して処理を進めていく
  - 先の例の短絡評価もその例の一つ
  - もっと言えば, `true && echo` は常に成功し, `true || echo` は常に失敗する
  - シェルスクリプトをマスターする上で最重要項目の一つ

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる
  - 全てのコマンドは基本的に終了ステータスを返す
  - 他の言語で言う戻り値に相当する
  - 終了ステータスが 0 か非 0 か, つまり真か偽かを判定して処理を進めていく
  - 先の例の短絡評価もその例の一つ
  - もっと言えば, `true && echo` は常に成功し, `true || echo` は常に失敗する
  - シェルスクリプトをマスターする上で最重要項目の一つ
  - すべてのコマンドがそうであるように, 自作コマンドやユーザ定義関数でも終了ステータスをしっかり返す設計であるべき

# 制御文と test コマンド

- 終了ステータス
  - 終了コード, Exit コード, ステータスコード, などとも呼ばれる
  - 全てのコマンドは基本的に終了ステータスを返す
  - 他の言語で言う戻り値に相当する
  - 終了ステータスが 0 か非 0 か, つまり真か偽かを判定して処理を進めていく
  - 先の例の短絡評価もその例の一つ
  - もっと言えば, `true && echo` は常に成功し, `true || echo` は常に失敗する
  - シェルスクリプトをマスターする上で最重要項目の一つ
  - すべてのコマンドがそうであるように, 自作コマンドやユーザ定義関数でも終了ステータスをしっかり返す設計であるべき
  - 0 ~ 255 の値を取るが, 1 以上はコマンドによって意味が異なるので, 気になる場合はその都度 `man` を引く(127 など決め打ちの値も存在する)

# 制御文と test コマンド

- if 文
- for 文
- while 文
- 真偽値
- 短絡評価
- 終了ステータス
- 数値と文字列の扱いの違い

# 制御文と test コマンド

- 数値と文字列の扱いの違い

# 制御文と test コマンド

- 数値と文字列の扱いの違い
  - 基本的にシェルでは違いがない(型なし)

# 制御文と test コマンド

- 数値と文字列の扱いの違い
  - 基本的にシェルでは違いがない(型なし)
  - 故に、すべて文字列として扱われる

# 制御文と test コマンド

- 数値と文字列の扱いの違い
  - 基本的にシェルでは違いがない(型なし)
  - 故に、すべて文字列として扱われる
  - `test 123 = 123` と `test 123 -eq 123` はどちらも真を返す

# 制御文と test コマンド

- 数値と文字列の扱いの違い
  - 基本的にシェルでは違いがない(型なし)
  - 故に、すべて文字列として扱われる
  - `test 123 = 123` と `test 123 -eq 123` はどちらも真を返す
  - 他の比較演算子でも同じ

# 制御文と test コマンド

- 数値と文字列の扱いの違い
  - 基本的にシェルでは違いがない(型なし)
  - 故に、すべて文字列として扱われる
  - `test 123 = 123` と `test 123 -eq 123` はどちらも真を返す
  - 他の比較演算子でも同じ
  - 同じ原理で、`test "123" -eq "123"` も真を返す  
(数値を文字列として比較することできる)

# シェルスクリプトの文法

- ・ シェルの基本機能 ✓
- ・ 制御文と test コマンド ✓
- ・ シェルの組み込みコマンド
- ・ 変数と関数

# シェルスクリプトの文法

- ・ シェルの基本機能 ✓
- ・ 制御文と test コマンド ✓
- ・ シェルの組み込みコマンド
- ・ 変数と関数



シェルの組み込みコマンド

# シェルの組み込みコマンド

- 今まで紹介してきたコマンドはほとんどがシェルの組み込みコマンドである

# シェルの組み込みコマンド

- 今まで紹介してきたコマンドはほとんどがシェルの組み込みコマンドである
- 一般のコマンド(外部コマンド)とは違って利点などがある

# シェルの組み込みコマンド

- ・ 内部コマンド

# シェルの組み込みコマンド

- 内部コマンド



- 環境に依存されない。シェルさえ同じであれば、他のマシンやアーキテクチャでも同じコマンドを使うことができる

# シェルの組み込みコマンド

- 内部コマンド
  -  環境に依存しない。シェルさえ同じであれば、他のマシンやアーキテクチャでも同じコマンドを使うことができる
  -  外部コマンドより実行速度が早いことが多い。外部コマンドは環境変数 \$PATH を探索したり、呼び出しまでに時間が掛かることがある

# シェルの組み込みコマンド

- 内部コマンド
  -  環境に依存しない。シェルさえ同じであれば、他のマシンやアーキテクチャでも同じコマンドを使うことができる
  -  外部コマンドより実行速度が早いことが多い。外部コマンドは環境変数 \$PATH を探索したり、呼び出しまでに時間が掛かることがある
  -  インタプリタが起動してさえいれば、いつでもどこでも起動できる

# シェルの組み込みコマンド

- 内部コマンド
  -  環境に依存しない。シェルさえ同じであれば、他のマシンやアーキテクチャでも同じコマンドを使うことができる
  -  外部コマンドより実行速度が早いことが多い。外部コマンドは環境変数 \$PATH を探索したり、呼び出しまでに時間が掛かることがある
  -  インタプリタが起動してさえいれば、いつでもどこでも起動できる
  -  基本的な機能しか提供されない

# シェルの組み込みコマンド

- ・ 外部コマンド

# シェルの組み込みコマンド

- 外部コマンド
  -  高機能なものが多い。ユーザが作ったものであったり、GNUなどの組織が作ったものであったりするため、シンプルであるが強力なコマンドである

# シェルの組み込みコマンド

- 外部コマンド
  -  高機能なものが多い。ユーザが作ったものであったり、GNUなどの組織が作ったものであったりするため、シンプルであるが強力なコマンドである
  -  原則カレントディレクトリでしか起動できない。カレントディレクトリ以外からも起動するには、環境変数 \$PATH の設定が必要

# シェルの組み込みコマンド

- 外部コマンド
  -  高機能なものが多い。ユーザが作ったものであったり、GNUなどの組織が作ったものであったりするため、シンプルであるが強力なコマンドである
  -  原則カレントディレクトリでしか起動できない。カレントディレクトリ以外からも起動するには、環境変数 \$PATH の設定が必要
  -  環境に依存する。例えば、tail コマンドの -r オプションは BSD 系に由来するため、GNU/Linux のそれには -r オプションはない(など)

# シェルの組み込みコマンド

- 外部コマンド

-  高機能なものが多い。ユーザが作ったものであったり、GNUなどの組織が作ったものであったりするため、シンプルであるが強力なコマンドである
-  原則カレントディレクトリでしか起動できない。カレントディレクトリ以外からも起動するには、環境変数 \$PATH の設定が必要
-  環境に依存する。例えば、tail コマンドの -r オプションは BSD 系に由来するため、GNU/Linux のそれには -r オプションはない(など)
- 外部コマンドは大きな差異まではないものの、些細な違いが Unix 系 OS には残っている。それは Unix の派閥が広く増え、独自の拡張や改良・開発がされてきた歴史的経緯に基づくものである。こうした違いを吸収するために、「最低限の統一」をはかる POSIX という規格が制定されている。

# シェルの組み込みコマンド

- 内部コマンド(ビルドインコマンド)には bash だけでもたくさんの種類があるため、今回は省略する
- 各自、手引書や検索エンジンで参照

# シェルの組み込みコマンド

- 内部コマンド(ビルドインコマンド)には bash だけでもたくさんの種類があるため、今回は省略する
- 各自、手引書や検索エンジンで参照
- ただ、シェルスクリプトを書くにあたって特筆すべきコマンドだけ紹介

# シェルの組み込みコマンド

- `exit` コマンド

# シェルの組み込みコマンド

- `exit` コマンド
  - 書式: `exit [ num ]`

# シェルの組み込みコマンド

- `exit` コマンド
  - 書式: `exit [ num ]`
  - 意味: `num` を終了コードとして現在のプロセスを終了する

# シェルの組み込みコマンド

- `exit` コマンド
  - 書式: `exit [ num ]`
  - 意味: `num` を終了コードとして現在のプロセスを終了する
  - `num` を省略した場合, 0 になる
  - 0 – 255 までの値しか設定できないが, それ以上を書くと, 256 で割った余りになる

# シェルの組み込みコマンド

- `return` コマンド

# シェルの組み込みコマンド

- `return` コマンド
  - 書式: `return [ num ]`
  - 意味: `num` を戻り値(終了コード)として関数を終了する

# シェルの組み込みコマンド

- `return` コマンド
  - 書式: `return [ num ]`
  - 意味: `num` を戻り値(終了コード)として関数を終了する
  - `num` を省略した場合, 0 になる
  - 0 – 255 までの値しか設定できないが, それ以上を書くと, 256 で割った余りになる

# シェルの組み込みコマンド

- `return` コマンド
  - 書式: `return [ num ]`
  - 意味: `num` を戻り値(終了コード)として関数を終了する
  - `num` を省略した場合, 0 になる
  - 0 – 255 までの値しか設定できないが, それ以上を書くと, 256 で割った余りになる

# シェルの組み込みコマンド

- その他にも exec, local, shift, kill, trap, continue, break, export, read, echo, type などがあるので各自チェック

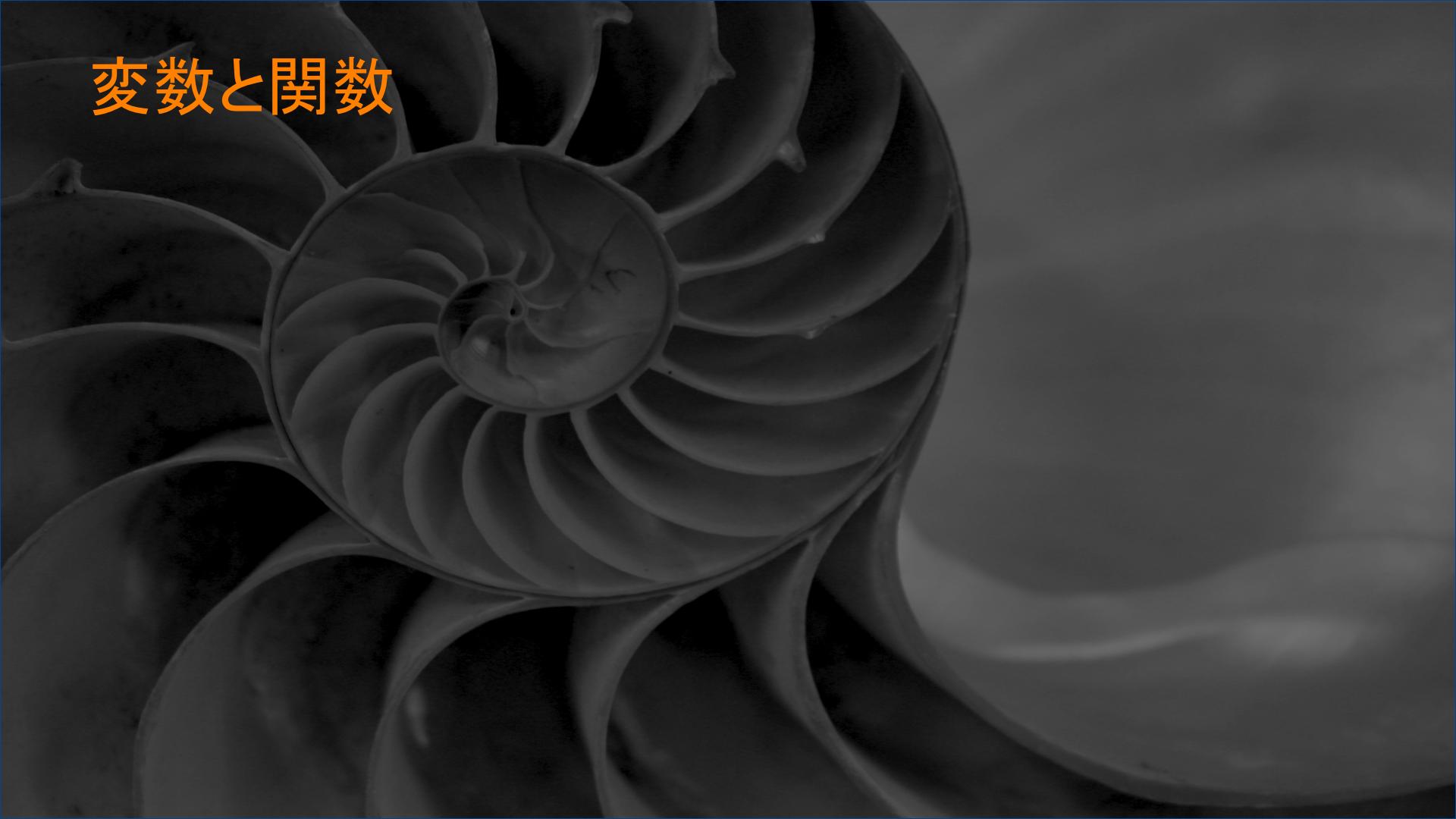
# シェルスクリプトの文法

- シェルの基本機能 ✓
- 制御文と test コマンド ✓
- シェルの組み込みコマンド ✓
- 変数と関数

# シェルスクリプトの文法

- ・ シェルの基本機能 ✓
- ・ 制御文と test コマンド ✓
- ・ シェルの組み込みコマンド ✓
- ・ 変数と関数

# 変数と関数



# 変数と関数

- 変数と関数

# 変数と関数

- 変数と関数
  - 変数 → シェルの基本機能

# 変数と関数

- 変数と関数
  - 変数 → シェルの基本機能
  - a=123
  - echo \$a

# 変数と関数

- 変数と関数

- 変数 → シェルの基本機能
- `a=123`
- `echo $a`
- 拡張的な変数展開がある

# 変数と関数

- 変数展開

# 変数と関数

- 変数展開

大文字小文字編集	
<code>\$ {PARAMETER^}</code>	先頭1文字を大文字化
<code>\$ {PARAMETER^^}</code>	すべての文字を大文字化
<code>\$ {PARAMETER, }</code>	先頭1文字を小文字化
<code>\$ {PARAMETER,, }</code>	すべての文字を小文字化
<code>\$ {PARAMETER~}</code>	先頭1文字を反転
<code>\$ {PARAMETER~~}</code>	すべての文字を反転

# 変数と関数

- 変数展開

部分文字列消去	
<code>\$ {PARAMETER#PATTERN}</code>	先頭の PATTERN を1つ消去
<code>\$ {PARAMETER##PATTERN}</code>	先頭の PATTERN をすべて消去
<code>\$ {PARAMETER%PATTERN}</code>	末尾の PATTERN を1つ消去
<code>\$ {PARAMETER%%PATTERN}</code>	末尾の PATTERN をすべて消去

# 変数と関数

- 変数展開

検索と置き換え	
<code> \${PARAMETER/PAT/STR}</code>	先頭の PAT を1つ STR に置き換える
<code> \${PARAMETER//PAT/STR}</code>	PAT すべてを STR に置き換える
<code> \${PARAMETER/PAT}</code>	先頭の PAT を1つ、空に置き換える
<code> \${PARAMETER//PAT}</code>	すべての PAT を空に置き換える

# 変数と関数

- 変数展開

変数の長さ	
<code>\$ {#PARAMETER}</code>	PARAMETER の長さを返す

部分文字列展開	
<code>\$ {PARAMETER:OFFSET}</code>	OFFSET から最後までを取り出す
<code>\$ {PARAMETER:OFFSET:LENGTH}</code>	OFFSET から LENGTH 分取り出す

# 変数と関数

- 変数展開

デフォルト値	
<code>\$ {PARAMETER:-WORD}</code>	空白か未定義のとき WORD を返す
<code>\$ {PARAMETER-WORD}</code>	未定義のとき WORD を返す
<code>\$ {PARAMETER:=WORD}</code>	空白か未定義のとき WORD を代入
<code>\$ {PARAMETER=WORD}</code>	未定義のとき WORD を代入

# 変数と関数

- 変数展開

オルタネート値	
<code>\$ {PARAMETER:+WORD}</code>	定義済みで非空白のとき WORD を返す
<code>\$ {PARAMETER+WORD}</code>	定義済みのとき WORD を返す
<code>\$ {PARAMETER:?WORD}</code>	空白か未定義のとき WORD でエラー表示
<code>\$ {PARAMETER?WORD}</code>	未定義のとき WORD でエラー表示

# 変数と関数

- 変数展開
  - 他にもたくさんあるが省略
  - 詳しくは, [Parameter expansion – Bash Hackers](#)

# 変数と関数

- 関数

# 変数と関数

- 関数
  - ユーザ定義関数を作れる

# 変数と関数

- ・ 関数
  - ユーザ定義関数を作れる

```
function name() {  
    处理  
}
```

# 変数と関数

- ・ 関数
  - ユーザ定義関数を作れる

```
function name() {  
    处理  
}
```

```
function name {  
    处理  
}
```

# 変数と関数

- ・ 関数
  - ユーザ定義関数を作れる

```
function name() {    name() {  
    处理                      处理  
}  
}
```

```
function name {  
    处理  
}
```

# 変数と関数

- ・ 関数
  - ユーザ定義関数を作れる

```
function name() {    name() {        function name()
  处理          处理          处理
}                }            {
}                  }            }
```

```
function name {
  处理
}
```

# 変数と関数

- ・ 関数
  - ユーザ定義関数を作れる

```
function name() {    name() {    function name()
  处理          处理          处理
}                }            {
}                  }            }

function name {
  处理
}

function name() { 处理; }
```

# 変数と関数

- ・ 関数
  - ユーザ定義関数を作れる

```
function name() {    name() {    function name()
  处理          处理          处理
}                }            {
}                }            }
function name {
  处理
}
function name() { 处理; }
```

- これ以外にも、これらを組み合わせた書きパターンがある

# 変数と関数

- ・ 関数
  - ユーザ定義関数を作れる

```
function name() {    name() {        function name()
  处理          处理          处理
}                }            {
}                  }
```

```
function name {
  处理
}
```

```
function name() { 处理; }
```

- これ以外にも、これらを組み合わせた書きパターンがある
- どれか一つを覚えて、最低限コード内では統一するべき

# 変数と関数

- 関数
  - ユーザ定義関数を作れる

```
function name() { 处理; }
```

- 関数は、最後に到達すると、自動で return 0 されるが、きちんとユーザが return ポイントを返り値をもって設定すべき

# 変数と関数

- 関数
  - ユーザ定義関数を作れる

```
function name() { 处理; }
```

- 関数は、最後に到達すると、自動で `return 0` されるが、きちんとユーザが `return` ポイントを返り値をもって設定すべき
- 正常終了なら 0、異常終了なら非 0 を返すようにする

# 変数と関数

- 関数
  - ユーザ定義関数を作れる

```
function name() { 处理; }
```

- 関数は、最後に到達すると、自動で `return 0` されるが、きちんとユーザが `return` ポイントを返り値をもって設定すべき
- 正常終了なら 0、異常終了なら非 0 を返すようにする
- こうすることで短絡評価や、条件文による分岐ができる

# 変数と関数

- 関数
  - ユーザ定義関数を作れる

```
function name() { 处理; }
```

- 関数は、最後に到達すると、自動で `return 0` されるが、きちんとユーザが `return` ポイントを返り値をもって設定すべき
- 正常終了なら 0、異常終了なら非 0 を返すようにする
- こうすることで短絡評価や、条件文による分岐ができる
- → ユーザ定義関数であっても内部コマンドと同じ扱いができる

# 変数と関数

- 関数の引数

# 変数と関数

- 関数の引数
  - `$1 - $9` で受け取る

# 変数と関数

- 関数の引数
  - `$1 - $9` で受け取る
- 引数の渡し方

# 変数と関数

- 関数の引数
  - \$1 - \$9 で受け取る
- 引数の渡し方

```
# contains returns true if the specified string contains
contains() {
    string="$1"
    substring="$2"
    if [ "${string#*$substring}" != "$string" ]; then
        return 0      # $substring is in $string
    else
        return 1      # $substring is not in $string
    fi
}
```

# シェルスクリプトの文法

- ・ シェルの基本機能 ✓
- ・ 制御文と test コマンド ✓
- ・ シェルの組み込みコマンド ✓
- ・ 変数と関数 ✓

# シェルスクリプトの文法

- シェルの基本機能 ✓
- 制御文と test コマンド ✓
- シェルの組み込みコマンド ✓
- 変数と関数 ✓
- 基本的な説明はこれでおしまい

# シェルスクリプトの文法

- シェルの基本機能 ✓
  - 制御文と test コマンド ✓
  - シェルの組み込みコマンド ✓
  - 変数と関数 ✓
- 
- 基本的な説明はこれでおしまい
  - 次は実践的なシェルスクリプトを書く技術について

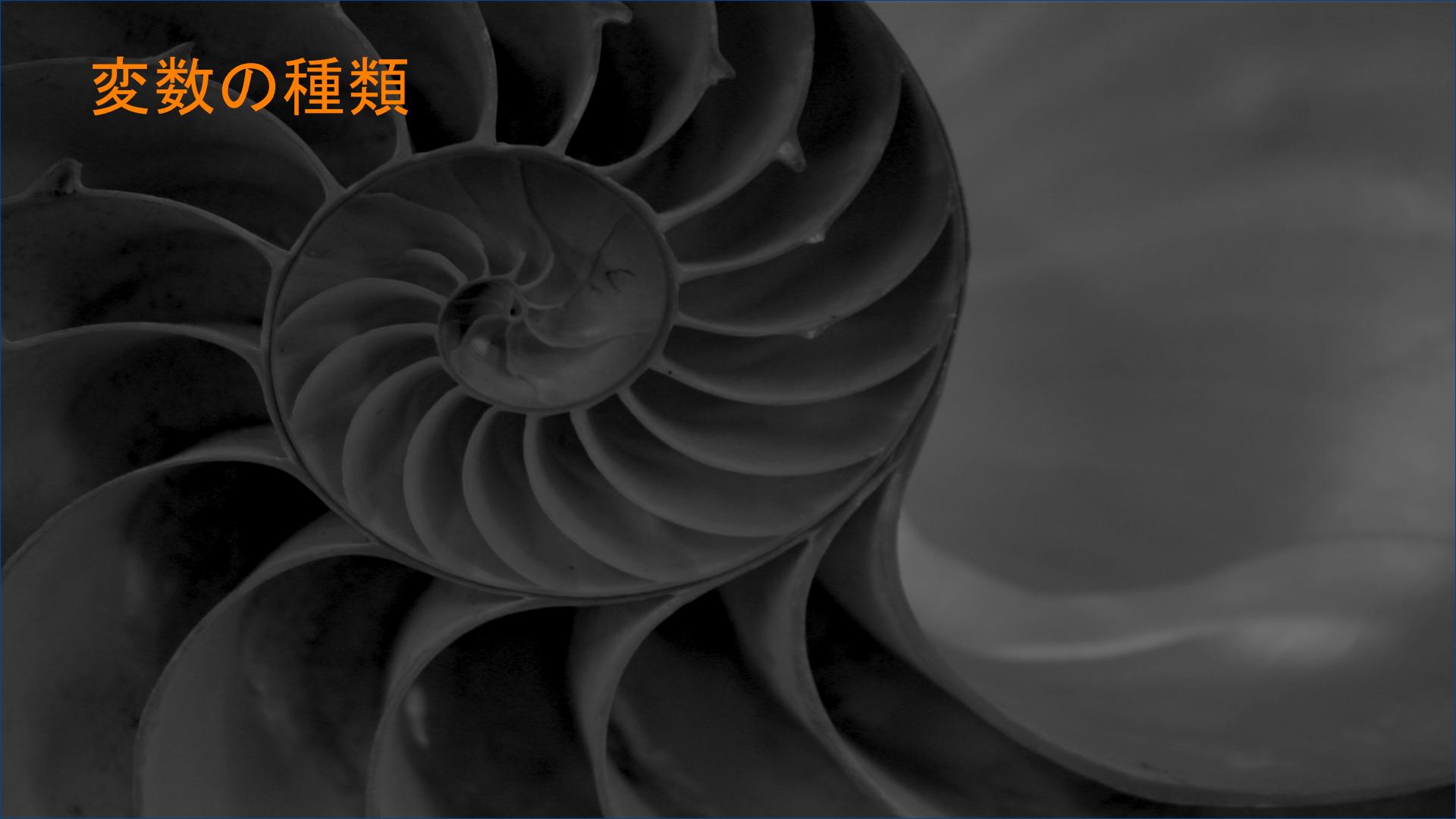
# シェルスクリプトの実践



# シェルスクリプトの実践

- 変数の種類
- コマンド置換
- プロセス置換
- 算術評価
- サブシェル
- POSIX

# 変数の種類



# 変数の種類

- シェルの変数には種類がある

# 変数の種類

- ・ シェルの変数には種類がある
  - 環境変数
  - シェル変数
  - 特殊変数

# 変数の種類

- ・ シェルの変数には種類がある
  - 環境変数
  - シェル変数
  - 特殊変数
- ・ スコープの概念は基本的にはない

# 変数の種類

- シェルの変数には種類がある
  - 環境変数
  - シェル変数
  - 特殊変数
- スコープの概念は基本的にはない
  - bash 以上のシェルに限って言えばローカル変数を作ることもできる

# 変数の種類

- 環境変数

# 変数の種類

- 環境変数
  - OS のデータ共有用に使われるシステム変数のこと。特に、タスクに対する外部からの設定を施すのに有効である

# 変数の種類

- 環境変数

- OS のデータ共有用に使われるシステム変数のこと。特に、タスクに対する外部からの設定を施すのに有効である
- わかりやすく：何らかの設定を使う。それは OS であったり、アプリケーションであったりする

# 変数の種類

- 環境変数
  - OS のデータ共有用に使われるシステム変数のこと. 特に, タスクに対する外部からの設定を施すのに有効である
  - わかりやすく: 何らかの設定を使う. それは OS であったり, アプリケーションであったりする
- 環境変数の参照

# 変数の種類

- 環境変数
  - OS のデータ共有用に使われるシステム変数のこと。特に、タスクに対する外部からの設定を施すのに有効である
  - わかりやすく：何らかの設定を使う。それは OS であったり、アプリケーションであったりする
- 環境変数の参照
  - 基本機能のときに触れた変数と変わらない
  - `echo $VARIABLE`

# 変数の種類

- 環境変数
  - OS のデータ共有用に使われるシステム変数のこと。特に、タスクに対する外部からの設定を施すのに有効である
  - わかりやすく：何らかの設定を使う。それは OS であったり、アプリケーションであったりする
- 環境変数の参照
  - 基本機能のときに触れた変数と変わらない
  - `echo $VARIABLE`
  - 環境変数は変数名がすべて大文字であることが望ましい

# 変数の種類

- ・ 環境変数の例

# 変数の種類

- 環境変数の例

- \$HOME
- \$PATH
- \$SHELL
- \$PWD
- \$EDITOR
- \$LANG

# 変数の種類

- 環境変数の例

- \$HOME ユーザのホームディレクトリのパスが設定される
- \$PATH コマンド検索パスを指定する。コロン区切りで複数指定できる
- \$SHELL 現在のシェルの起動パスが設定される
- \$PWD カレントディレクトリが設定される
- \$EDITOR エディタのコマンド名を指定する
- \$LANG ロケールを指定する

# 変数の種類

- 環境変数の例
  - \$HOME ユーザのホームディレクトリのパスが設定される
  - \$PATH コマンド検索パスを指定する。コロン区切りで複数指定できる
  - \$SHELL 現在のシェルの起動パスが設定される
  - \$PWD カレントディレクトリが設定される
  - \$EDITOR エディタのコマンド名を指定する
  - \$LANG ロケールを指定する
- 環境変数の宣言

# 変数の種類

- 環境変数の例
  - \$HOME ユーザのホームディレクトリのパスが設定される
  - \$PATH コマンド検索パスを指定する。コロン区切りで複数指定できる
  - \$SHELL 現在のシェルの起動パスが設定される
  - \$PWD カレントディレクトリが設定される
  - \$EDITOR エディタのコマンド名を指定する
  - \$LANG ロケールを指定する
- 環境変数の宣言
  - VARIABLE="value"; export VARIABLE
  - export VARIABLE="value"

# 変数の種類

- 環境変数の例
  - \$HOME ユーザのホームディレクトリのパスが設定される
  - \$PATH コマンド検索パスを指定する。コロン区切りで複数指定できる
  - \$SHELL 現在のシェルの起動パスが設定される
  - \$PWD カレントディレクトリが設定される
  - \$EDITOR エディタのコマンド名を指定する
  - \$LANG ロケールを指定する
- 環境変数の宣言
  - VARIABLE="value"; export VARIABLE
  - export VARIABLE="value"
  - いずれか。下記は bash によるものだったが、最新の POSIX によるとどちらも可

# 変数の種類

- 環境変数の例
  - \$HOME ユーザのホームディレクトリのパスが設定される
  - \$PATH コマンド検索パスを指定する。コロン区切りで複数指定できる
  - \$SHELL 現在のシェルの起動パスが設定される
  - \$PWD カレントディレクトリが設定される
  - \$EDITOR エディタのコマンド名を指定する
  - \$LANG ロケールを指定する
- 環境変数の宣言
  - VARIABLE="value"; export VARIABLE
  - export VARIABLE="value"
  - いずれか。下記は bash によるものだったが、最新の POSIX によるとどちらも可

# 変数の種類

- ・ シェル変数

# 変数の種類

- シェル変数
  - カレントシェル内で宣言された変数. 一般的な変数

# 変数の種類

- シェル変数
  - カレントシェル内で宣言された変数. 一般的な変数
- シェル変数の参照
  - `echo $variable`

# 変数の種類

- シェル変数
  - カレントシェル内で宣言された変数. 一般的な変数
- シェル変数の参照
  - `echo $variable`
- シェル変数の宣言
  - `variable="value"`

# 変数の種類

- ・ シェル変数のスコープ

# 変数の種類

- シェル変数のスコープ
  - 基本的にグローバル変数として扱われる

# 変数の種類

- シェル変数のスコープ
  - 基本的にグローバル変数として扱われる
  - ただし、カレントシェル内のみで有効

# 変数の種類

- シェル変数のスコープ
  - 基本的にグローバル変数として扱われる
  - ただし、カレントシェル内のみで有効
  - ある関数内で宣言されたシェル変数も大域的に扱われる

# 変数の種類

- シェル変数のスコープ
  - 基本的にグローバル変数として扱われる
  - ただし、カレントシェル内のみで有効
  - ある関数内で宣言されたシェル変数も大域的に扱われる
  - つまり関数の外からアクセスできる

# 変数の種類

- シェル変数のスコープ
  - 基本的にグローバル変数として扱われる
  - ただし、カレントシェル内のみで有効
  - ある関数内で宣言されたシェル変数も大域的に扱われる
  - つまり関数の外からアクセスできる
  - しかし、内部コマンドの `local` 修飾子で関数ローカルな変数を作り出すことができる

# 変数の種類

- ・ シェル変数のスコープ
  - 基本的にグローバル変数として扱われる
  - ただし、カレントシェル内のみで有効
  - ある関数内で宣言されたシェル変数も大域的に扱われる
  - つまり関数の外からアクセスできる
  - しかし、内部コマンドの `local` 修飾子で関数ローカルな変数を作り出すことができる

```
function some() {  
    local a b  
    a="$1"  
    b="$2"  
    local c="$a$b"  
    echo "$c"  
}  
some hoge fuga  
echo $c
```

# 変数の種類

- ・ シェル変数のスコープ
  - 基本的にグローバル変数として扱われる
  - ただし、カレントシェル内のみで有効
  - ある関数内で宣言されたシェル変数も大域的に扱われる
  - つまり関数の外からアクセスできる
  - しかし、内部コマンドの `local` 修飾子で関数ローカルな変数を作り出すことができる
  - `local` がない場合、関数外から `c` にアクセスできるが、この場合はアクセスできず、ブランクが表示される

```
function some() {  
    local a b  
    a="$1"  
    b="$2"  
    local c="$a$b"  
    echo "$c"  
}  
some hoge fuga  
echo $c
```

# 変数の種類

- ・ 環境変数とシェル変数の違い

# 変数の種類

- 環境変数とシェル変数の違い
  - 間違えやすいポイントのひとつ
  - 環境変数はすべて大文字で、エクスポート(外部に公開)された変数

# 変数の種類

- 環境変数とシェル変数の違い
  - 間違えやすいポイントのひとつ

# 変数の種類

- 環境変数とシェル変数の違い
  - 間違えやすいポイントのひとつ
  - 環境変数はすべて大文字で、エクスポート(外部に公開)された変数
  - シェル変数はエクスポートされていない変数

# 変数の種類

- 環境変数とシェル変数の違い
  - 間違えやすいポイントのひとつ
  - 環境変数はすべて大文字で、エクスポート(外部に公開)された変数
  - シェル変数はエクスポートされていない変数
  - どちらもシェルの振る舞いを変える

# 変数の種類

- 環境変数とシェル変数の違い
  - 間違えやすいポイントのひとつ
  - 環境変数はすべて大文字で、エクスポート(外部に公開)された変数
  - シェル変数はエクスポートされていない変数
  - どちらもシェルの振る舞いを変える

# 変数の種類

- 環境変数とシェル変数の違い
  - 間違えやすいポイントのひとつ
  - 環境変数はすべて大文字で、エクスポート(外部に公開)された変数
  - シェル変数はエクスポートされていない変数
  - どちらもシェルの振る舞いを変える
  - 環境変数はプロセス間をまたぐ(親プロセスから子プロセスに引き継がれる)

# 変数の種類

- 環境変数とシェル変数の違い
  - 間違えやすいポイントのひとつ
  - 環境変数はすべて大文字で、エクスポート(外部に公開)された変数
  - シェル変数はエクスポートされていない変数
  - どちらもシェルの振る舞いを変える
  - 環境変数はプロセス間をまたぐ(親プロセスから子プロセスに引き継がれる)
  - 環境変数は超大域的な変数、シェル変数は大域的な変数、ローカルなシェル変数は局所的な変数

# 変数の種類

- 環境変数とシェル変数の違い
  - 間違えやすいポイントのひとつ
  - 環境変数はすべて大文字で、エクスポート(外部に公開)された変数
  - シェル変数はエクスポートされていない変数
  - どちらもシェルの振る舞いを変える
  - 環境変数はプロセス間をまたぐ(親プロセスから子プロセスに引き継がれる)
  - 環境変数は超大域的な変数、シェル変数は大域的な変数、ローカルなシェル変数は局所的な変数

**DEMO**

# 変数の種類

- 特殊変数

# 変数の種類

- 特殊変数
  - 特殊な組み込みシェル変数

# 変数の種類

- ・ 特殊変数
  - 特殊な組み込みシェル変数

変数名	値
\$0	スクリプト名
\$1, \$2, ..., \$9	スクリプトや関数で、指定された引数(数字は位置)
\$#	引数の数
\$*	引数全て(" \$1 \$2 ")
\$@	引数全て(" \$1 " " \$2 ")
\$?	直前の終了ステータス

# まとめ

- ・ 大方のシェルスクリプトの文法を見てきた
- ・ これ以外にもコマンド置換や、プロセス置換などマストの文法がある
- ・ 配列や特殊変数など
- ・ 詳しくは、[Man page of BASH](#)