Rust クイックリファレンス

所有権、型、トレイト、パターンマッチングの基本

基本

Hello World

```
fn main() {
    println!("Hello, World!");
}
```

Cargo コマンド

```
cargo new my_project    # create new project
cargo build             # compile (debug)
cargo build --release   # compile (optimized)
cargo run               # build and run
cargo test              # run tests
```

プロジェクト構造

Cargo.toml	プロジェクトマニフェスト（依存関係、メタデータ）
src/main.rs	バイナリクレートのエントリーポイント
src/lib.rs	ライブラリクレートのルート
tests/	統合テストディレクトリ

変数と可変性

束縛と可変性

```
let x = 5;              // immutable by default
let mut y = 10;         // mutable
y += 1;
const MAX: u32 = 100;   // compile-time constant
```

シャドウイング

```
let x = 5;
let x = x + 1;          // shadows previous x
let x = "now a string"; // can change type
```

スカラー型

i8..i128, isize	符号付き整数
u8..u128, usize	符号なし整数
f32, f64	浮動小数点数（デフォルト f64）
bool	**true** / **false**
char	Unicode スカラー値（4 バイト）

複合型

```
let tup: (i32, f64, char) = (42, 6.4, 'z');
let (a, b, c) = tup;        // destructure
let arr: [i32; 3] = [1, 2, 3];
let first = arr[0];
```

関数

定義

```
fn add(a: i32, b: i32) -> i32 {
    a + b   // no semicolon = return expression
}
```

クロージャ

```
let double = |x: i32| x * 2;
let sum: i32 = vec![1, 2, 3]
    .iter()
    .map(|x| x * 2)
    .sum();
```

関数ポインタとトレイト

fn(T) -> U	関数ポインタ型
Fn(T) -> U	借用するクロージャ
FnMut(T) -> U	可変借用するクロージャ
FnOnce(T) -> U	所有権を取るクロージャ

制御フロー

If / Else

```
let status = if score >= 90 { "A" }
             else if score >= 80 { "B" }
             else { "C" };  // if is an expression
```

ループ

```
loop { break; }                 // infinite
while condition { }             // while
for item in &vec { }            // iterator
for i in 0..10 { }             // range
for (i, v) in vec.iter().enumerate() { }
```

ループラベル

```
'outer: for i in 0..5 {
    for j in 0..5 {
        if i + j > 6 { break 'outer; }
    }
}
```

所有権と借用

所有権のルール

1. 各値にはちょうど 1 つの所有者がいる。
2. 所有者がスコープを外れると値はドロップされる。
3. 値はムーブまたはクローンできる。

ムーブとクローン

```
let s1 = String::from("hello");
let s2 = s1;          // s1 is moved, no longer valid
let s3 = s2.clone();  // deep copy, both valid
```

借用

```
fn len(s: &String) -> usize { s.len() }  // shared ref
fn push(s: &mut String) { s.push('!'); }  // mutable ref
// Rule: many &T OR one &mut T, never both
```

ライフタイム

```
fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() { a } else { b }
}
```

構造体と列挙型

構造体

```
struct User {
    name: String,
    age: u32,
    active: bool,
}
let u = User { name: String::from("Alice"), age: 30, active: true };
```

impl ブロック

```
impl User {
    fn new(name: &str, age: u32) -> Self {
        Self { name: name.to_string(), age, active: true }
    }
    fn greeting(&self) -> String {
        format!("Hi, {}", self.name)
    }
}
```

列挙型

```
enum Shape {
    Circle(f64),
    Rect { w: f64, h: f64 },
    Point,
}
let s = Shape::Circle(5.0);
```

パターンマッチング

match 式

```
match shape {
    Shape::Circle(r) => std::f64::consts::PI * r * r,
    Shape::Rect { w, h } => w * h,
    Shape::Point => 0.0,
}
```

If Let と While Let

```
if let Some(val) = optional {
    println!("{val}");
}
while let Some(top) = stack.pop() {
    println!("{top}");
}
```

パターン構文

_	ワイルドカード、何にでもマッチ
x @ 1..=5	マッチした範囲を **x** に束縛
(a, b, ..)	タプルを分解、残りを無視
Some(x) if x > 0	マッチガード
Foo { x, .. }	構造体、他のフィールドを無視

エラー処理

Result と Option

```
enum Result<T, E> { Ok(T), Err(E) }
enum Option<T>    { Some(T), None }
```

?演算子

```
fn read_file(path: &str) -> Result<String, io::Error> {
    let mut s = String::new();
    File::open(path)?.read_to_string(&mut s)?;
    Ok(s)
}
```

エラーの処理

```
match result {
    Ok(val) => println!("{val}"),
    Err(e) => eprintln!("Error: {e}"),
}
let val = result.unwrap_or(0);
let val = result.unwrap_or_else(|_| default());
```

Rust クイックリファレンス

よく使うメソッド

`.unwrap()`	値を取得またはパニック
`.expect(msg)`	値を取得またはメッセージ付きでパニック
`.unwrap_or(default)`	値を取得またはデフォルトを使用
`.map(f)`	Ok/Some の値を変換
`.and_then(f)`	操作を連鎖（フラットマップ）
`.is_ok() / .is_some()`	真偽チェック

トレイト

定義と実装

```
trait Summary {
    fn summarize(&self) -> String;
    fn preview(&self) -> String {   // default impl
        format!("{}...", &self.summarize()[..20])
    }
}
impl Summary for User {
    fn summarize(&self) -> String { self.name.clone() }
}
```

トレイト境界

```
fn notify(item: &impl Summary) { }
fn notify<T: Summary + Display>(item: &T) { }
fn notify(item: &(impl Summary + Display)) { }
```

よく使うトレイト

`Display`	ユーザー向けの文字列フォーマット
`Debug`	デバッグフォーマット（`{:?}`）
`Clone, Copy`	複製（ディープ / ビット単位）
`PartialEq, Eq`	等価比較
`PartialOrd, Ord`	順序比較
`Iterator`	反復のための `next()`
`From, Into`	型変換
`Default`	デフォルト値のコンストラクタ

コレクション

Vec

```
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);
v.pop();              // returns Option<i32>
let first = &v[0];   // panics if empty
let first = v.get(0); // returns Option<&i32>
```

HashMap

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("key", 42);
m.entry("key").or_insert(0);
if let Some(val) = m.get("key") { }
```

String

```
let s = String::from("hello");
let s = "hello".to_string();
let combined = format!("{} {}", s, "world");
for c in s.chars() { }   // iterate characters
```

イテレータ

```
let sum: i32 = vec![1, 2, 3].iter().sum();
let doubled: Vec<_> = v.iter().map(|x| x * 2).collect();
let evens: Vec<_> = v.iter().filter(|x| *x % 2 == 0).collect();
```

並行処理

スレッド

```
use std::thread;
let handle = thread::spawn(|| {
    println!("from spawned thread");
});
handle.join().unwrap();
```

チャンネル

```
use std::sync::mpsc;
let (tx, rx) = mpsc::channel();
tx.send(42).unwrap();
let val = rx.recv().unwrap();
```

共有状態

`Arc<T>`	アトミック参照カウント（スレッドセーフな Rc）
`Mutex<T>`	相互排他、ロックして内部値にアクセス
`RwLock<T>`	複数の読み取りまたは 1 つの書き込み
`Send`	トレイト：スレッド間の転送が安全
`Sync`	トレイト：スレッド間の参照共有が安全

マクロと属性

よく使うマクロ

`println!()`	改行付きで出力
`format!()`	フォーマットされた String を返す
`vec![]`	リテラルから Vec を作成
`todo!()`	プレースホルダー、実行時にパニック
`assert!(expr)`	expr が false ならパニック
`assert_eq!(a, b)`	a != b ならパニック

Derive 属性

```
#[derive(Debug, Clone, PartialEq)]
struct Point { x: f64, y: f64 }
// Auto-implements Debug, Clone, PartialEq
```

テスト属性

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() { assert_eq!(add(2, 2), 4); }
    #[test]
    #[should_panic]
    fn it_panics() { panic!("boom"); }
}
```