

Rust速習会2

～Rustの苦いところハンズオン～

2018-10-01 @Wantedlyオフィス (白金台)

原 将己 (@qnighy)

実況用ハッシュタグ: #rust_jp

今日の予定

- 前半: Rust特有のハマりどころに焦点を当てて説明します。
 - Rustのコードを見てみよう
 - Rustのコスト感を知る
 - まずは型合わせから
 - 所有権とムーブ
 - ライフタイムと排他制御
 - 継承可変性と排他制御
 - 内部可変性と排他制御
 - 内部可変性とスレッド安全性
 - クロージャが難しくて……
 - ライフタイム落ち穂拾い

今日の予定

- 後半: ハンズオンをやります。
- 練習問題を解きつつ、設計まわりのハマりどころを説明します。
 - 関数シグネチャの設計
 - 型とトレイト
 - データ構造の設計

注意事項

- Rustの文法や型などの基本事項で、他のプログラミング言語からの類推が効く部分までは説明できていません。
- 「基本的な部分かも」と思っても、わからなかったら質問してもらえると助かります。他の人の助けにもなると思います。
- 自分で調べるときは、[TRPL](#) から近いセクションを探して読むか、[リファレンス](#) を見るといいでしょう。
- また、[APIリファレンス](#) は各自で参照してください。
 - リンクを貼る余裕がありませんでした。
- その他、資料については前回のスライドも参照

Rustのコードを見てみよう

Rustのコードを見てみよう：概要

- 実際にISUCON8で書いたコードを試みる。
- RubyよりRustのほうが数倍長い。
- Rustが本質的に長い部分もあるし、非同期まわりが未成熟なために長くなっている部分もある。
- この章はなんとなく雰囲気掴めればOK

Rustプログラミングの傾向

書きやすさ

- コードは長くなる傾向にある。
- コンパイルエラー駆動で書けるので、注意力は要らない

読みやすさ

- 長いので一覧性は高くない。
- コードの相互作用が明確化されており、追跡しやすい

どれくらい長くなるか……

- ISUCON8で作ったコードで例示します。
- 色々な要因
 - 本質的に必要な部分
 - Rustの非同期まわりが未成熟なための冗長性
 - Actix-Webが未成熟なための冗長性

参考: ISUCON8のコード片 (Ruby)

```
def get_login_user
  user_id = session[:user_id]
  return unless user_id
  @login_user ||= db.xquery('SELECT id, nickname FROM users WHERE id = ?',
user_id).first
end

get '/' do
  @user = get_login_user
  @events = get_events.map(&method(:sanitize_event))
  erb :index
end
```

参考: ISUCON8のコード片 (Rust 1)

```
#[derive(Debug, Serialize)]
struct UserWithNickname {
    id: u32,
    nickname: String,
}

#[derive(Debug, Template)]
#[template(path = "index.html")]
struct IndexTemplate {
    base_url: String,
    user: Option<UserWithNickname>,
    events: Vec<Event>,
}
```

参考: ISUCON8のコード片 (Rust 1)

```
#[derive(Debug, Serialize)]  
struct UserWithNickname {  
    id: u32,  
    nickname: String,  
}
```

JSONシリアライズと
テンプレートパラメーター用の構造体

```
#[derive(Debug, Template)]  
#[template(path = "index.html")]  
struct IndexTemplate {  
    base_url: String,  
    user: Option<UserWithNickname>,  
    events: Vec<Event>,  
}
```

テンプレート用構造体

参考: ISUCON8のコード片 (Rust 1)

```
#[derive(Debug, Serialize)]
struct UserWithNickname {
    id: u32,
    nickname: String,
}

#[derive(Debug, Template)]
#[template(path = "index.html")]
struct IndexTemplate {
    base_url: String,
    user: Option<UserWithNickname>,
    events: Vec<Event>,
}
```

derive(Debug):

デバッグ出力用の処理をコード生成している
(自分で定義することも可能)

derive(Serialize):

serdeライブラリのシリアライズ用処理

derive(Template):

Askamaライブラリ用

参考: ISUCON8のコード片 (Rust 1)

```
#[derive(Debug, Serialize)]
struct UserWithNickname {
    id: u32,
    nickname: String,
}

#[derive(Debug, Template)]
#[template(path = "index.html")]
struct IndexTemplate {
    base_url: String,
    user: Option<UserWithNickname>,
    events: Vec<Event>,
}
```

userは None になることがある

参考: ISUCON8のコード片 (Rust 2)

```
fn get_index(  
    (state, req, session): (State<TorbState>, HttpRequest<TorbState>, Session),  
) -> Result<impl Responder, Error> {  
    let base_url = {  
        if let Some(host) = req.headers().get("Host") {  
            format!("http://{host}", host.to_str().unwrap())  
        } else {  
            format!("{}", state.get("base_url").unwrap())  
        }  
    };  
    // ...  
}
```

参考: ISUCON8のコード片 (Rust 2)

```
fn get_index(  
    (state, req, session): (State<TorbState>, HttpRequest<TorbState>, Session),  
) -> Result<impl Responder, Error> {  
    let base_url = {  
        if let Some(host) = req.headers.get("Host") {  
            format!("http://{host}")  
        } else {  
            format!("{}", req.uri().host())  
        }  
    };  
    // ...  
}
```

リクエストに関する情報は必要なものだけ受け取る。
State はサーバー全体で共有する状態 (設定もココ)
HttpRequest と Session は名前通り

参考: ISUCON8のコード片 (Rust 2)

```
fn get_index(  
    (state, req, session): (State<TorbState>, HttpRequest<TorbState>, Session),  
) -> Result<impl Handler, Error> {  
    let base_url = {  
        if let Some(base_url) = req.get("Host") {  
            base_url  
        } else {  
            "http://localhost:8080/"  
        }  
    };  
    // ...  
}
```

普通は
state: State<TorbState>, req: HttpRequest<TorbState>,
session: Session
のように書く。
ハンドラのインターフェースを揃えるために1引数に押し込んでいる

参考: ISUCON8のコード片 (Rust 2)

```
fn get_index(  
    (state, req, session): (State<TorbState>, HttpRequest<TorbState>, Session),  
) -> Result<impl Responder, Error> {  
    let base_url = {  
        if let Some(host) = req.headers().get("Host") {  
            format!("http://{}/", req.uri().unwrap())  
        } else {  
            format!("{}", req.uri().unwrap())  
        }  
    };  
    // ...  
}
```

`Result<T, E>` はエラー処理用の汎用的な型。
E にはドメインごとの共通エラー型を使うことが多い
(ここでは `actix_web::Error` を使っている)

参考: ISUCON8のコード片 (Rust 2)

```
fn get_index(
    (state, req, session): (State<TorhState> HttpRequest<TorhState>
    Session),
) -> Result<impl Responder,
    let base_url = {
        if let Some(host) = req.headers().get("Host") {
            format!("http://{host}", host.to_str().unwrap())
        } else {
            format!("{}", ...)
        }
    };
    // ...
}
```

HostヘッダーからURLを生成する仕組みがなかったので自作

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql_pool.get().expect("mysql connection  
timeout");  
        // ...  
    });  
    // ...  
}
```

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql::Conn::new().expect("mysql connection  
timeout");  
        // ...  
    });  
    // ...  
}
```

Actix-Webのセッションは文字列のマップになっている。
数値等にパースして取り出すところまでやってくれる。

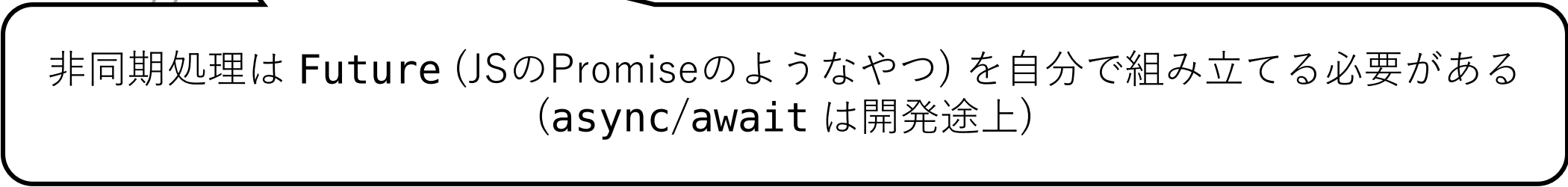
参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql_pool.get().expect("connection  
timeout");  
        // ...  
    });  
    // ...  
}
```

Result<Option<u32>, Error> が返ってくる。
? 演算子により、エラーを上に戻している。

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql_pool.get().expect("mysql connection  
        timeout");  
        //  
    });  
}
```



非同期処理は **Future** (JSのPromiseのようなやつ) を自分で組み立てる必要がある
(**async/await** は開発途上)

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql_pool.get().expect("mysql connection  
timeout");  
        //  
    });  
    // ...  
}
```

非同期DBアクセスはまだあまり整備されていないので、
ここは同期処理を使っている。
同期処理を別のスレッドにオフロードすることでブロッキングを防ぐ。

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql_pool.get().expect("mysql connection  
timeout");  
        //  
    });  
    // ...  
}
```

このクロージャは `get_index` 関数終了後に呼ばれる可能性がある。
そのため、ローカル変数を普通に参照するとコンパイルエラーになる。
`move` をつけてムーブキャプチャーにすることで回避できる。

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql_pool.get().expect("mysql connection  
timeout");  
        // ...  
    });  
    // ...  
}
```

ここでは、型推論をいい感じにするために戻り値型を明示している。

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql_pool.get().expect("mysql connection  
timeout");  
        // ...  
    });  
    // ...  
}
```

.expect するとエラーをパニックに変換できる。
(ここは ? にしておくべきだった感)

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let mysql_pool = state.mysql_pool.clone();  
    let user_id = session.get:::<u32>("user_id")?;  
    let fut = state.mysql_pool.spawn_fn(move || -> Result<_, Error> {  
        let mut conn = mysql_pool.get().expect("mysql connection  
timeout");  
        // ...  
    });  
    // ...  
}
```

コネクションプール(への **Arc** 参照) をここで一度複製している。
State は別スレッドに送れないため、これをしないと
コンパイルエラーになる。

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> { // ...
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> { // ...
        let user = if let Some(user_id) = user_id {
            conn.first_exec("SELECT nickname FROM users WHERE id = ?",
                (user_id,))
                .map_err(error::InternalServerError)?
                .map(|(nickname,)| UserWithNickname {
                    id: user_id,
                    nickname,
                })
        } else {
            None
        }; // ...
    }); // ...
}
```

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> { // ...
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> { // ...
        let user = if let Some(user_id) = user_id {
            conn.first_row("SELECT nickname FROM users WHERE id = ?",
                (user_id,))
                .await
                .map(|row| row.get(0))
                .ok()
        } else {
            None
        };
        match user {
            Some(nickname) => {
                Ok(Response::builder().body(nickname).build())
            }
            None => {
                Err(Error::NotFound)
            }
        }
    });
}
```

user_id は Option<u32> だった。
 ここでその場合分けを書いている。(match の構文糖衣)

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> { // ...
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> { // ...
        let user = if let Some(user_id) = user_id {
            conn.first_exec("SELECT nickname FROM users WHERE id = ?",
                (user_id,))
                .map_err(|_| Error::NotFound)
                .map(|row| {
                    let id = row.get(0);
                    let nickname = row.get(1);
                })
        } else {
            None
        }; // ...
    }); // ...
}
```

user_id をシャドウイングしたので、
if の中では user_id: u32 になる。

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> { // ...  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> { // ...  
        let user = if let Some(user_id) = user_id {  
            conn.first_exec("SELECT nickname FROM users WHERE id = ?",  
(user_id,))  
                .map_err(error::InternalServerError)?  
                .map(|(nickname,)| User { Nickname {  
                    id: user_id,  
                    nickname: nickname,  
                }})  
        } else {  
            None  
        }; // ...  
    }); // ...  
}
```

MySQLのエラーをActix-Webのエラーに自動変換できない。
ここでは汎用500エラーに明示的に変換している。

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> { // ...
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> { // ...
        let user = if let Some(user_id) = user_id {
            conn.first_exec("SELECT nickname FROM users WHERE id = ?",
                (user_id,))
                .map_err(error::InternalServerError)?
                .map(|(nickname,)| UserWithNickname {
                    id: user_id,
                    name: nickname,
                })
        } else {
            None
        }
    }); // ...
}
```

first_exec なので Option<行> が返ってくる。
今回は Option::map を使って処理する。

参考: ISUCON8のコード片 (Rust 3)

```
fn get_index(...) -> Result<impl Responder, Error> { // ...
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> { // ...
        let user = if let Some(user_id) = user_id {
            conn.first_exec("SELECT nickname FROM users WHERE id = ?",
                (user_id,))
                .map_err(error::InternalServerError)?
                .map(|(nickname,)| UserWithNickname {
                    id: user_id,
                    nickname: nickname,
                })
        } else {
            Err(error::NotFound)
        }
    }); // ...
}; // ...
}
```

行は複数個のフィールドからなり、今回はたまたま1要素だった。
そのため、1要素のタプルが返ってくる。
1要素のタプルでは末尾カンマが必須 (Python と同じ)

参考: ISUCON8のコード片 (Rust 4)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        // ...  
        let events = get_events(&mut conn, true)?;  
  
        Ok((user, events))  
    });  
    // ...  
}
```

参考: ISUCON8のコード片 (Rust 4)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {  
        // ...  
        let events = get_events(&mut conn, true)?;  
  
        Ok((user, events))  
    });  
    // ...  
}
```

user, eventsだけ取り出してスレッドプールを抜ける。
コネクションはRAIIでいい感じに返却される

参考: ISUCON8のコード片 (Rust 5)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let fut = fut.map(|(user, events)| IndexTemplate {  
        base_url,  
        user,  
        events,  
    });  
    Ok(Box::new(fut) as Box<Future<Item = _, Error = Error>>)  
}
```

参考: ISUCON8のコード片 (Rust 5)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let fut = fut.map(|(user, events)| IndexTemplate {  
        base_url,  
        use_events, ...  
    });  
    Ok(Box::new(fut) as Box<Future<Item = _, Error = Error>>)  
}
```

さっき作った fut に map して処理を繋げている

参考: ISUCON8のコード片 (Rust 5)

```
fn get_index(...) -> Result<impl Responder, Error> {  
    // ...  
    let fut = fut.map(|(user, events)| IndexTemplate {  
        base_url,  
        user,  
        events,  
    });  
    Ok(Response::from_fut(fut))  
}
```

セッションへの書き込みとかはスレッドプールに出せないなのでここでやる。
(Actixがシングルスレッドのため)
indexの場合は分ける必然性はない

参考: ISUCON8のコード片 (Rust 5)

```
fn get_index(...) -> Result<impl Responder, Error> {
```

```
// ...  
let
```

Actix-Webの **Responder** にあわせるために **Box** に入れる

```
events,  
});
```

```
Ok(Box::new(fut) as Box<Future<Item = _, Error = Error>>)
```

```
}
```

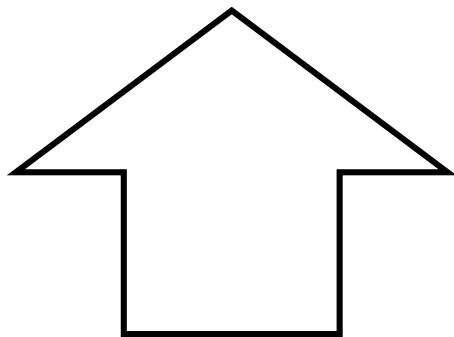
えっ……長くない？

- えっ……長くない？

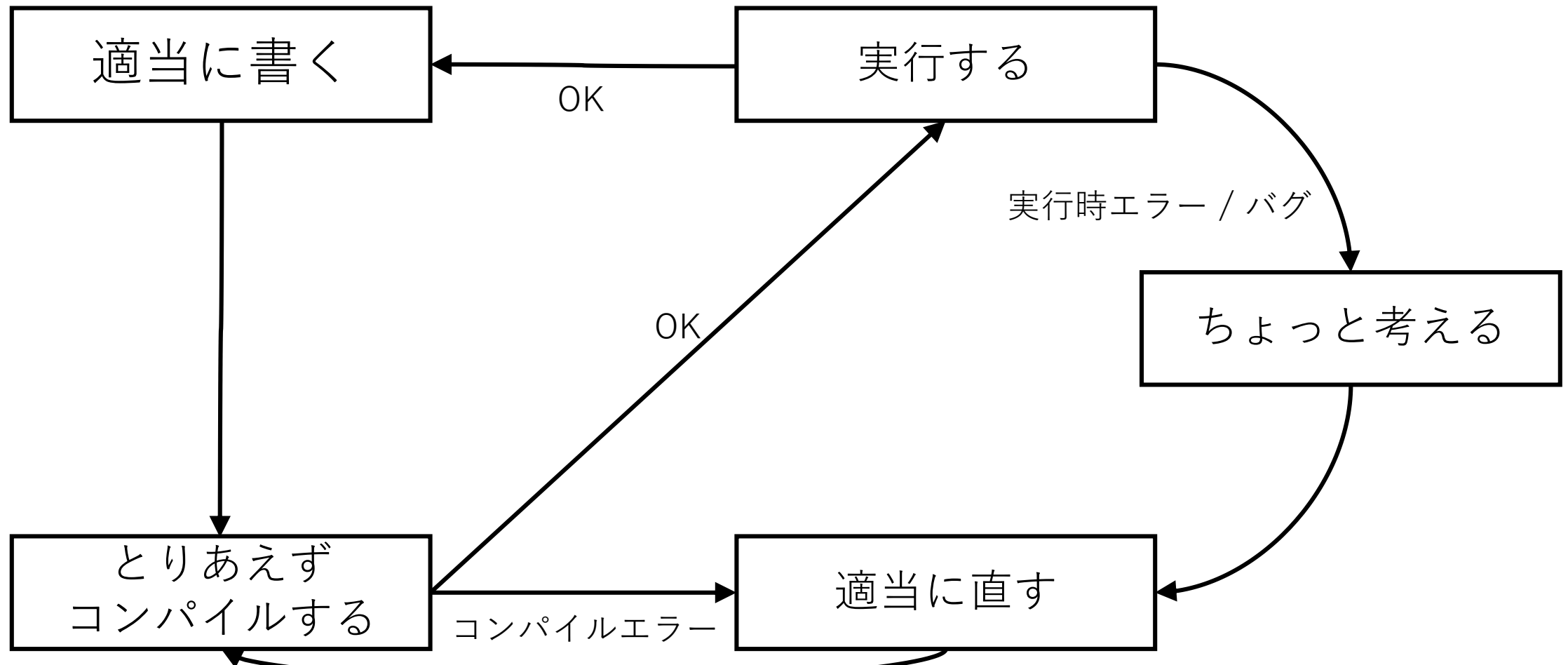
えっ……長くない？

参加要件

- Mac, Windows, Linuxなど、Rust処理系が動作する環境を用意してください。 [Tier1サポートのあるプラットフォーム](#)が望ましいです。
- コンパイラを信じ、エラーメッセージと向き合う気持ち



コンパイルエラー駆動開発！



エラーが親切な例

ムーブキャプチャーの必要があるとき

```
error[E0373]: closure may outlive the current function, but it borrows `mysql_pool`, which is owned by the current function
--> src/main.rs:89:39

89 |     let fut = state.cpu_pool.spawn_fn(|| -> Result<_, Error> {
    |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^ may outlive borrowed value `mysql_pool`
90 |         let mut conn = mysql_pool.get().expect("mysql connection timeout");
    |                     ----- `mysql_pool` is borrowed here
help: to force the closure to take ownership of `mysql_pool` (and any other referenced variables), use the `move` keyword
89 |     let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {
    |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error[E0373]: closure may outlive the current function, but it borrows `user_id`, which is owned by the current function
--> src/main.rs:89:39

89 |     let fut = state.cpu_pool.spawn_fn(|| -> Result<_, Error> {
    |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^ may outlive borrowed value `user_id`
...
92 |         let user = if let Some(user_id) = user_id {
    |                     ----- `user_id` is borrowed here
help: to force the closure to take ownership of `user_id` (and any other referenced variables), use the `move` keyword
89 |     let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {
    |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

ちょっと考える必要がある例

Send じゃないものを他スレッドに送ろうとした

```
error[E0277]: `std::rc::Rc<TorbState>` cannot be sent between threads safely
--> src/main.rs:88:30
88 |         let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {
    |                                     ^^^^^^^ `std::rc::Rc<TorbState>` cannot be sent between threads safely
    |
    = help: within `[closure@src/main.rs:88:39: 105:6 state:actix_web::State<TorbState>, user_id:std::option::Option<u32>]`, the trait `std::marker::Send` is not implemented for `std::rc::Rc<TorbState>`
    = note: required because it appears within the type `actix_web::HttpRequest<TorbState>`
    = note: required because it appears within the type `actix_web::State<TorbState>`
    = note: required because it appears within the type `[closure@src/main.rs:88:39: 105:6 state:actix_web::State<TorbState>, user_id:std::option::Option<u32>]`

error[E0277]: `std::rc::Rc<actix_web::router::ResourceMap>` cannot be sent between threads safely
--> src/main.rs:88:30
88 |         let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {
    |                                     ^^^^^^^ `std::rc::Rc<actix_web::router::ResourceMap>` cannot be sent between threads safely
    |
    = help: within `[closure@src/main.rs:88:39: 105:6 state:actix_web::State<TorbState>, user_id:std::option::Option<u32>]`, the trait `std::marker::Send` is not implemented for `std::rc::Rc<actix_web::router::ResourceMap>`
    = note: required because it appears within the type `actix_web::dev::ResourceInfo`
    = note: required because it appears within the type `actix_web::HttpRequest<TorbState>`
    = note: required because it appears within the type `actix_web::State<TorbState>`
    = note: required because it appears within the type `[closure@src/main.rs:88:39: 105:6 state:actix_web::State<TorbState>, user_id:std::option::Option<u32>]`
```

ちょっと考える必要がある例

Send じゃないものを他スレッドに送ろうとした

```
error[E0277]: `std::rc::Rc<TorbState>` cannot be sent between threads safely
--> src/main.rs:88:30
88 |     let fut = state.cpu_pool.spawn_fn(move || -> Result<_, Error> {
    |                                     ^^^^^^^ `std::rc::Rc<TorbState>` cannot be sent between threads safely
```

- Rc はスレッド非セーフな参照カウンタ
- Rc なんて使ってないぞ……？
- →よく見るとActix-Webの内部で使われている
- Actix-Webの **State** はスレッドプールに持ち込めない
ので必要な部分だけ取り出して送る必要があるんだ
な！

ちょっと考える必要がある例

最後に **Box** する必要があるのにしていない

```
error[E0277]: the trait bound `tokio::prelude::future::Map<futures_cpupool::CpuFuture<(std::option::Option<UserWithNickname>, std::vec::Vec<Event>), actix_web::Error>, [closure@src/main.rs:108:23: 112:6 base_url:_]>: actix_web::Responder` is not satisfied
--> src/main.rs:78:13
78 | ) -> Result<impl Responder, Error> {
   |           ^^^^^^^^^^^^^^^^^ the trait `actix_web::Responder` is not implemented for `tokio::prelude::future::Map<futures_cpupool::CpuFuture<(std::option::Option<UserWithNickname>, std::vec::Vec<Event>), actix_web::Error>, [closure@src/main.rs:108:23: 112:6 base_url:_]>`
   |
   = note: the return type of a function must have a statically known size
```

↑ とんちんかんなnoteが出ている

ちょっと考える必要がある例

最後に **Box** する必要があるのにしていない

```
error[E0277]: the trait bound `tokio::prelude::future::Map<futures_cpupool::CpuFuture<(std::option::Option<UserWithNickname>, std::vec::Vec<Event>), actix_web::Error>, [closure@src/main.rs:108:23: 112:6 base_url:_]>: actix_web::Responder` is not satisfied
```

- **Responder** は非同期でもいいはずなので **Future** を取れないのはおかしい
 - 仮説1: **Item/Error** が一致していない？
 - 仮説2: 実は **Future** に対して実装されていない？
- **Responder** のドキュメントを読むと、 **Box<dyn Future>** にしか実装されていないことが判明

コンパイルエラー駆動とは言うけれど

- はまりがちな罫は先にわかっておいたほうがよい
- 順番に紹介します！

Rustのコスト感を知る

ライフタイムに入る前に、コスト感を知っておきましょう

Rustのコストを知る: 概要

- 実行時コストがRustの設計に深く関わっている、という感覚を掴む。
- 実行時コストのかかるものがオプティンになっている傾向がある。

Rustのコスト感

```
x = [1, 2, 3]
```

このPythonコードをRustに翻訳すると……

Rustのコスト感

```
let mut x =  
    Arc::new(Mutex::new(  
        vec![1, 2, 3]));
```

だいたいこんな感じになる。
普通はここまで重装備である必要はない

Rustのコスト感

```
let mut x =  
    Arc::new(Mutex::new(  
        vec![1, 2, 3]));
```

Mutex は動的に排他制御する。
→ いらなら外す

Rustのコスト感

```
let mut x =  
    Arc::new(vec![1, 2, 3]);
```

Arc は生存期間を動的に制御する。
→ いらないうちで外す

Rustのコスト感

```
let mut x = vec![1, 2, 3];
```

Vec は動的に長さを変えられる。
→ いらないうえ外す

Rustのコスト感

```
let mut x = [1, 2, 3];
```

`mut` は書き換え可能であることを示す
→ いらないうら外す
(`mut`に実行時コストはない)

Rustのコスト感

```
let x = [1, 2, 3];
```

Rustでは軽いほうがデフォルト。
必要になってから実行時コストを
オプトインしていく。

Rustのコスト感

```
let x = [1, 2, 3];
```

言語が実行時コストを教えてくれる
→ 普段から気にする必要はない

まずは型合わせから

まずは型合わせから: 概要

- はまりがちなコンパイルエラーのうち、単純な型違いを埋める。
- **Option, Result**, 参照型のつけ外しの方法を知る。

Option/Result と和解せよ

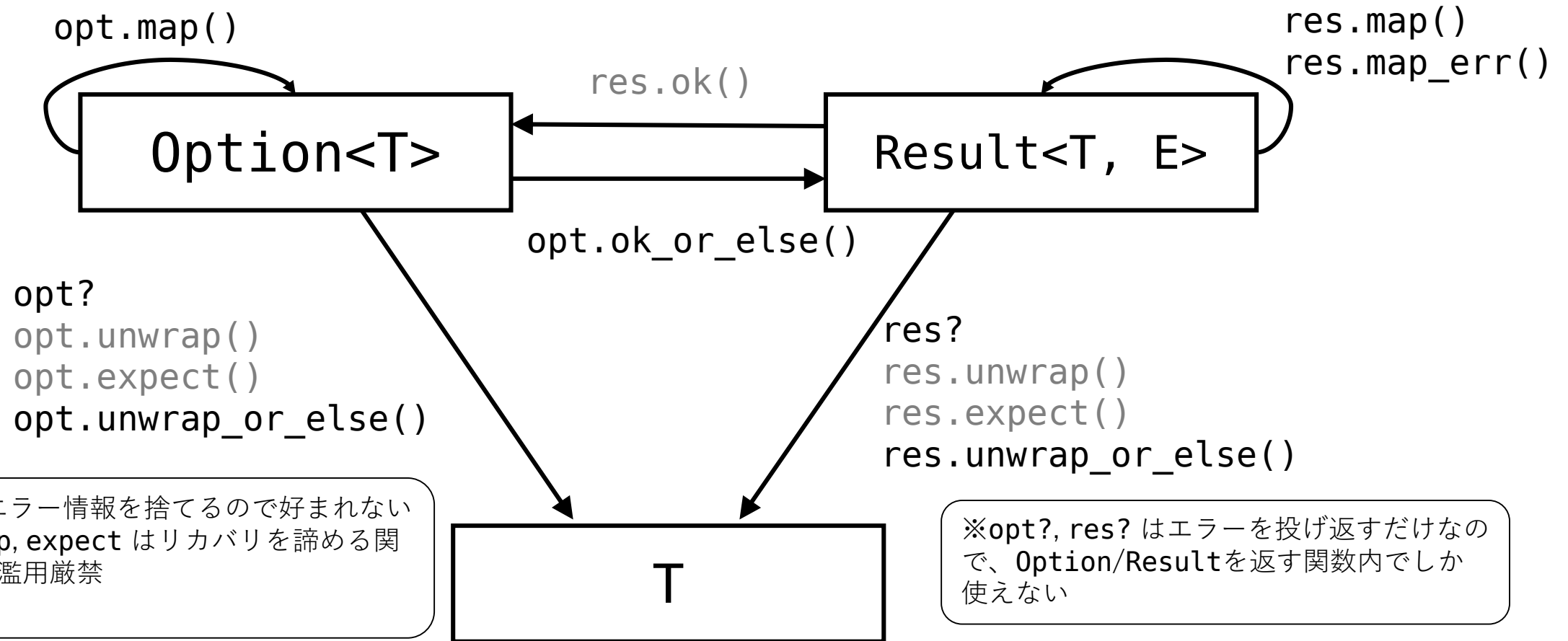
- ML系関数型言語ではおなじみの概念
- `Option<T>`: `Some(t)`, `None` のどちらか
- `Result<T, E>`: `Ok(t)`, `Err(e)` のどちらか

※汎用のEitherについてはeither crateを参照

- 付属のメソッドを使ってもいいし、自力でパターンマッチを書いてもOK

※`Option/Result`の内容に応じて`continue/break/return`したいときは自力でパターンマッチするのが定石

Option/Result と和解せよ



参照型の大原則

- `&T`, `&mut T` は独立した型であり、型合わせのときは他の型と同じように振る舞う
- たとえば……
 - `Option<&'a T>` と `&'a Option<T>` は別の型である。
 - `Option<&'a T>` と `Option<&'a mut T>` は別の型である。
 - `Option<&'a &'b T>` と `Option<&'a T>` は別の型である。

この原則に対する例外はよく見られる。

- 自動参照外し
- 自動参照 (特にメソッド記法のレシーバー)
- 型強制と再借用

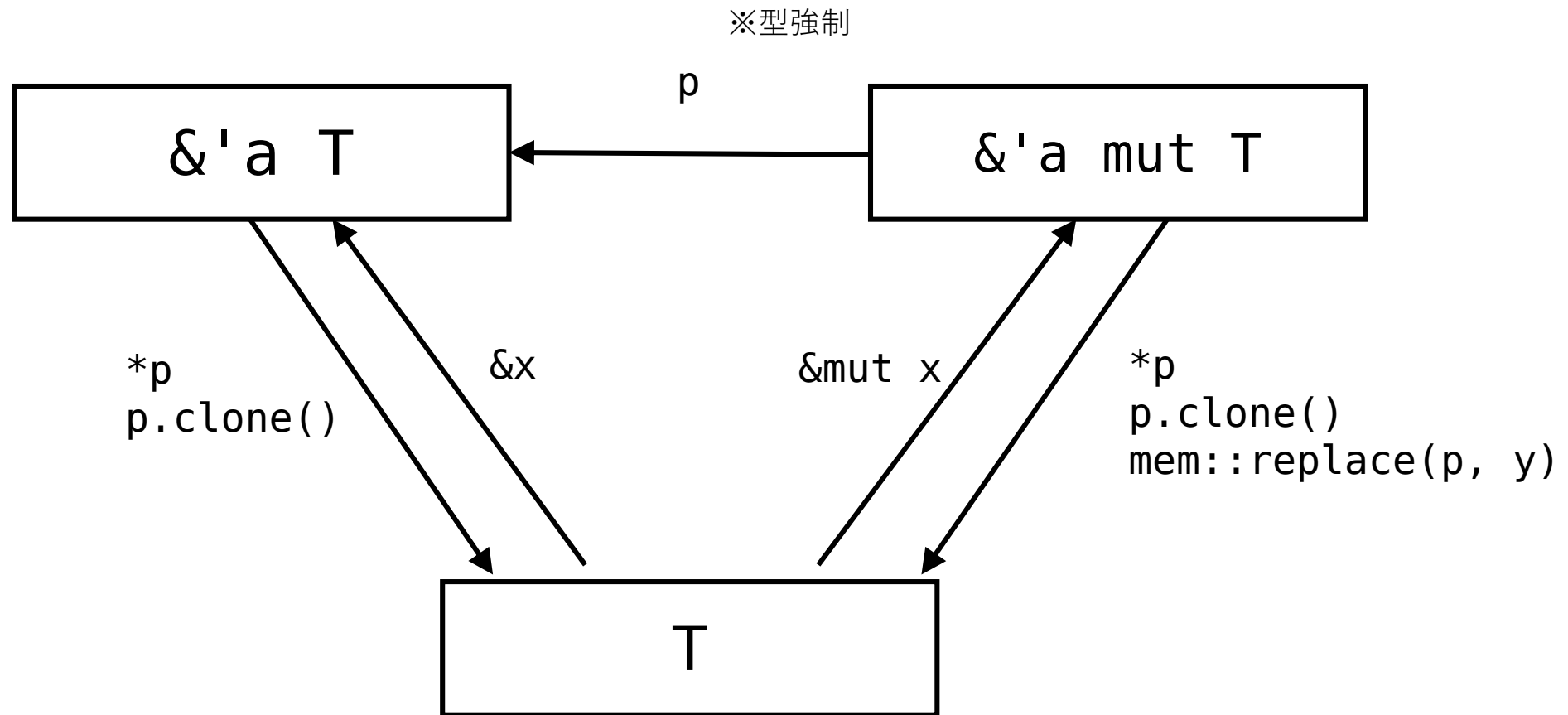
値と参照の三つ巴

- `&T`, `&mut T`, `T` の三つ巴（または最初の2つ）で並立しているAPIは結構ある

たとえば……

- `Fn`, `FnMut`, `FnOnce`
- `Deref`, `DerefMut`, `--`
- `Index`, `IndexMut`, `--`
- `Borrow`, `BorrowMut`, `--`
- `AsRef`, `AsMut`, `Into`
- `iter()`, `iter_mut()`, `into_iter()`
- `split_at()`, `split_at_mut()`, `--`
- 式のモード（イミュータブル左辺値、ミュータブル左辺値、右辺値）
- パターンのモード（イミュータブル左辺値、ミュータブル左辺値、右辺値）

値と参照の三つ巴



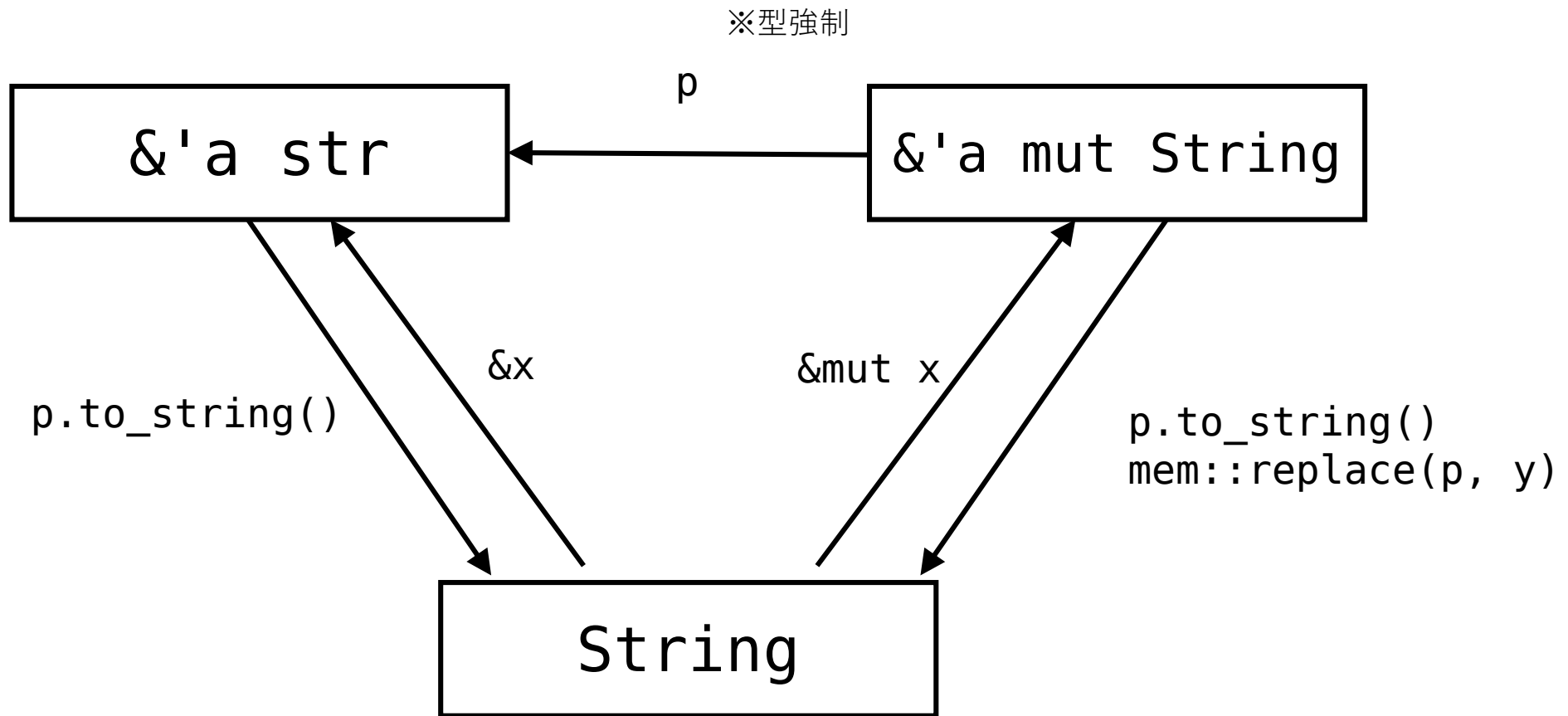
値と参照の三つ巴 (unsized版)

- 配列、文字列、トレイトオブジェクトの三つ巴は少し異なる

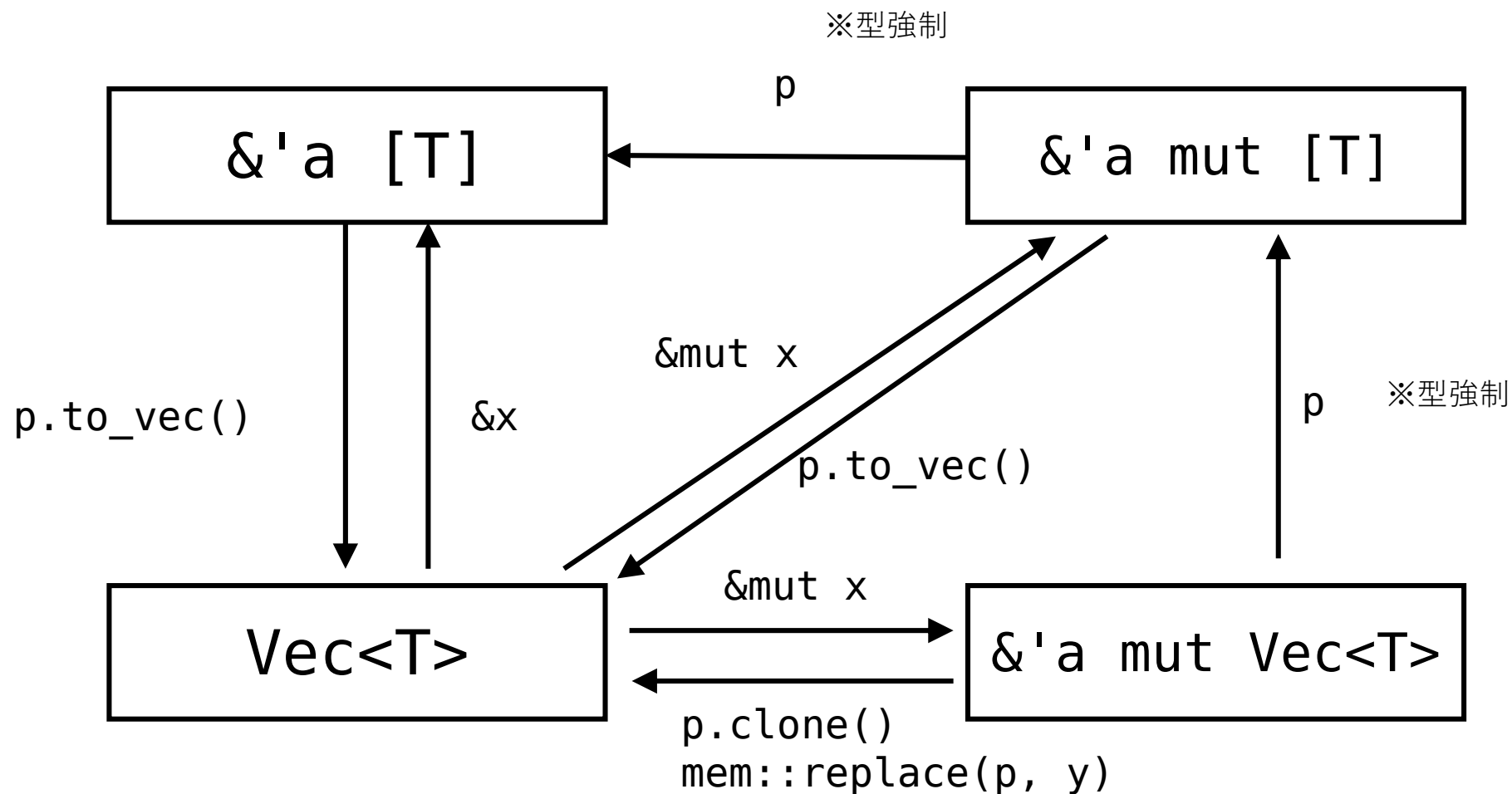
基本形	<code>&'a T</code>	<code>&'a mut T</code>	<code>T</code>	
トレイトオブジェクト	<code>&'b dyn Tr + 'a</code>	<code>&'b mut dyn Tr + 'a</code>	<code>Box<dyn Tr + 'a></code>	
配列	<code>&'a [T]</code>	<code>&'a mut [T]</code>	<code>Box<[T]></code>	<code>[T; n]</code>
	<code>&'a Vec<T></code>	<code>&'a mut Vec<T></code>	<code>Vec<T></code>	
文字列	<code>&'a str</code>	<code>&'a mut str</code>	<code>Box<str></code>	
	<code>&'a String</code>	<code>&'a mut String</code>	<code>String</code>	

- Unsized typeはポインタ経由でしか扱えないので、**Box** で包む (fat pointer になる)
- **Vec<T>**, **String** は可変長な **Box** である。(長さ と 容量 を 別々に 持つ)
- 長さフィールドをケチる理由があまりないため **Box<[T]>**, **Box<str>** はあまり使われない。
- UTF-8文字列をバイト数を変えずに置き換える需要がないため、**&mut str** はほとんど使われない。
- イミュータブル参照では可変長である意味はないので **&Vec<T>**, **&String** は使わない。
- **[T; n]** はコンパイル時に長さが決まる (文字列に対応する型はない)

値と参照の三つ巴 (文字列版)



値と参照の三つ巴(?) (配列版)



参照を見落としがちなところ

- Q. x の型はなんでしょう？

```
let v: Vec<i32> = vec![1, 2, 3];  
for x in v.iter() {  
    // ...  
}
```

参照を見落としがちなところ

- Q. x の型はなんでしょう？

```
let v: Vec<SomeBigTree> = /* ... */;  
for x in v.iter() {  
    // ...  
}
```

「とっても大きな木」
のように、複製にコストがかかるものだったら……？

参照を見落としがちなところ

- Q. x の型はなんでしょう？

```
let v: Vec<Mutex<Conn>> = /* ... */;  
for x in v.iter() {  
    // ...  
}
```

「mutexガードされたコネクション」
のように、複製できないものだったら……？

参照を見落としがちなところ

- Q. x の型はなんでしょう？

```
let v: Vec<i32> = vec![1, 2, 3];  
for x in v.iter() {
```

ここには「要素への参照」が入ってくる！

参照を見落としがちなところ

- Q. x の型はなんでしょう？

```
let v: Vec<i32> = vec![1, 2, 3];  
for x in v.iter() {
```

ちなみに、要素への参照ではなく要素で受けるには……

- for x in x.iter().cloned() とする (要素型が **Clone** のとき使える)
- for &x in v.iter() とする (要素型が **Copy** のとき使える)
- for x in v.drain(..) とする (いつでも使える。配列は空になる)
- for x in v (あるいは v.into_iter()) とする (いつでも使える。配列ごと手放すことになる)

秘技！自動参照外し

```
let s: String = ...;  
let s = s.trim();
```

この s は &str 型になっている！

秘技！自動参照外し

```
let s: String = ...;  
let s = s.trim();
```

このようなパターンはよく見られるが、実際の仕組みは以下の通り。

- `s.trim()` のようなメソッド記法はドットを使わない方法で置き換えられる。
 - 今回は `str::trim` の糖衣。
- この第一引数は `s`, `&s`, `&mut s` のいずれかの形が採用される。
 - 今回は `str::trim(&s)` となる。
- `&` を使った位置では任意個の `*` がついたものとみなされる。
 - 今回は `str::trim(&*s)` となる。
- `*x` はオーバーロード可能で `*Deref::deref(&s)` に脱糖される。
 - `str::trim(&*String::deref(&s))` となる。

参照を見落としがちなところ

- Q. x の型はなんでしょう？

```
let v: Vec<i32> = vec![1, 2, 3];  
for x in v.iter() {  
    //  
}
```

この v も実は &v を渡している！

所有権とムーブ

所有権とムーブ: 概要

- 所有権は何らかの唯一性を保障するのに使える仕組みで、その内実は初期化チェックの逆にすぎない。
- Rustにおけるムーブは、所有権が絡む以外はコピーとほぼ同じである。

所有権

- 問題: 同じものが**複数あると困る**ことがある（詳しくは後述）
- 解答: **未初期化チェック**を応用しよう

Javaの未初期化チェック

- コンパイル時に検査される

```
int x;  
System.out.println("x = " + x);
```

未初期化！

Rustの未初期化チェック

- Rustも同じ

```
let x: u32;  
println!("{}", x);
```

未初期化！

Rustのムーブ検査

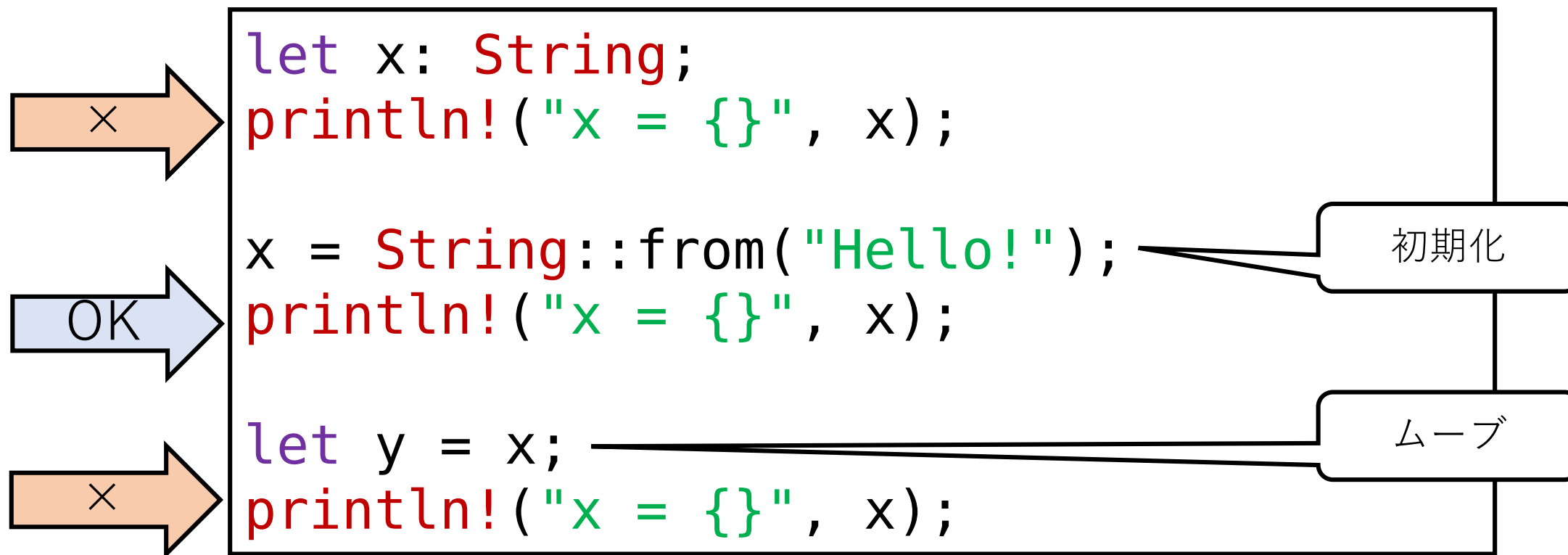
- よくある最小例

```
let x = String::from("Hello!");  
let y = x;  
println!("x = {}", x);
```

ムーブ済みなのでエラー！

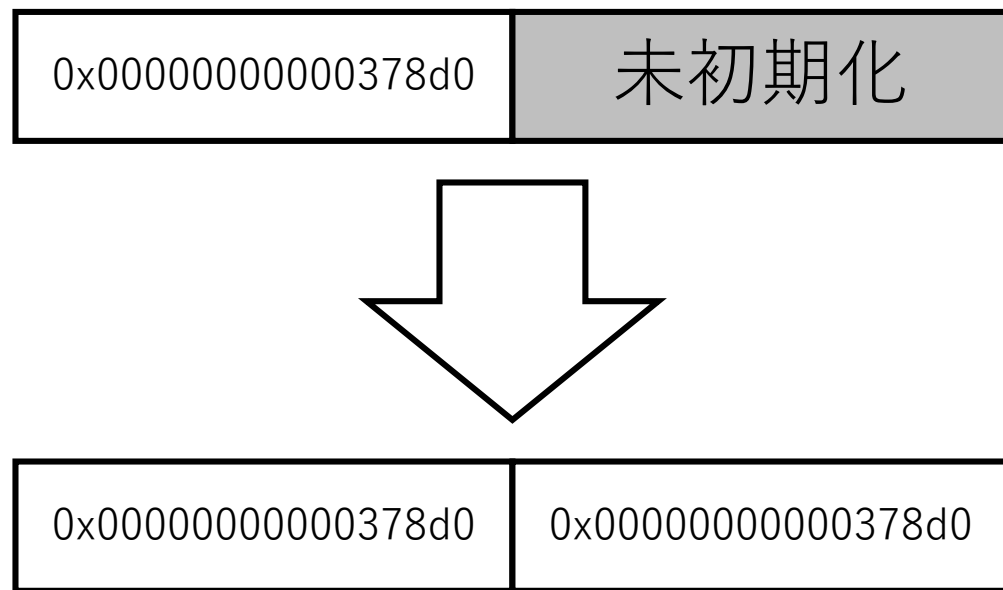
Rustのムーブ検査

- 初期化とムーブアウトは逆の操作



ムーブはほとんどコピー

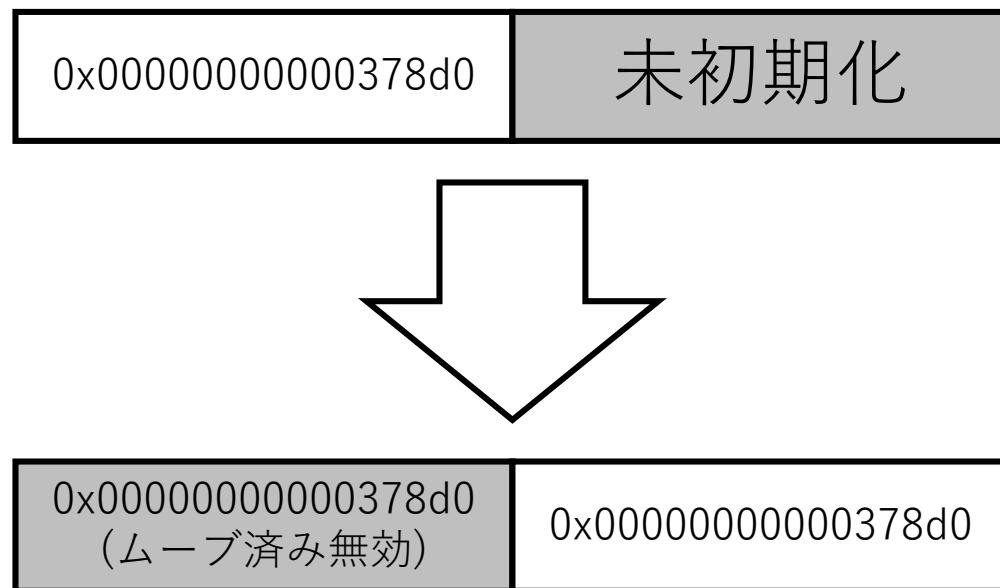
- Rustの「コピー」は**単純コピー**



※Rust用語では、単純コピーをコピーと呼び、複雑な処理が必要なものはクローンと呼ばれる。
これはC++の用語法とは異なる。

ムーブはほとんどコピー

- Rustの「ムーブ」も**単純コピー**



※C++のムーブは単純コピーではない。
※条件付きムーブなどではdrop flagが生成される場合がある。
その場合は単純コピーに加えて、フラグを立てる処理が加わる。

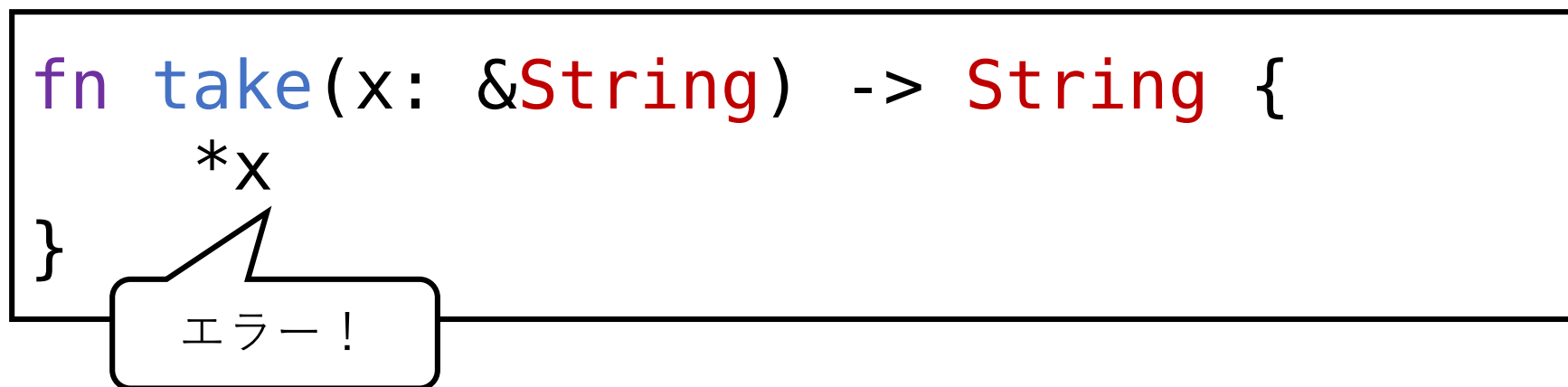
参照からはムーブできない

- 参照からコピーはできる

```
fn take(x: &i32) -> i32 {  
    *x  
}
```

参照からはムーブできない

- 参照から**ムーブ**はできない




コピーになるか、ムーブになるかは組み込みの `Copy` トレイトで制御されている。

参照からはムーブできない

- mut参照でもダメ！

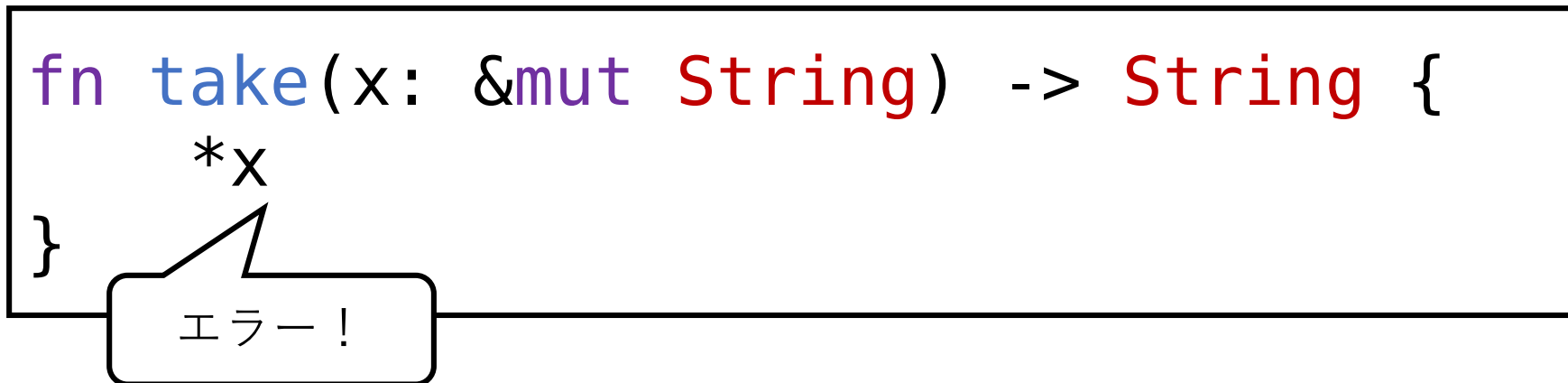
```
fn take(x: &mut String) -> String {  
    *x  
}
```



- 参照は中身を入れて返さないといけないため、空にできない

参照からはムーブできない

- mut参照でもダメ！



- mut参照の場合、代わりの中身を入れることでムーブできる。そのためには標準ライブラリの `replace` 関数を使う。初期化/未初期化の状態遷移が複雑なときは、動的な制御にオプトすることもできる。そのためには `Option` を使い、`None` を未初期化、`Some` を初期化済みとみなす。`Option::replace` や `Option::take` メソッドを補助として用いることができる。
この目的で `Option` が必要になるのは、`Future` の実装など特殊なライブラリが多い。アプリケーションコードでこのパターンが必要になることはあまり多くないと思われる。

困ったらクローン

- クローンはディープコピーを行う（Arc/Rc に対しては浅いコピー）

```
fn take(x: &String) -> String {  
    x.clone()  
}
```

困ったらクローン

- ハンドル系の値はクローンできないこともある

```
fn take(x: &File) -> File {  
    x.clone()  
}
```

エラー！

上記のコードをコンパイルすると、本来あるべきエラー(cloneできない)ではなく、型エラーが表示される。これはメソッド解決のルールが悪さをしている。**File** がクローンできないが、**&File** はクローンできるため、そちらだと思ってコンパイルが続行された結果、型エラーになる。
また、**File** は通常の意味ではクローンできないが、**dup** 系のシステムコールで複製する **try_clone** がある。

困ったらクローン

- Arc/Rc に包めば浅いコピーになるため、クローンできる

```
fn take(x: &Arc<File>) -> Arc<File> {  
    x.clone()  
}
```

OK! 同じハンドルを指している

Arc/Rc に包むときは、内部可変性が一緒に必要になることが多い。(後述)
ライブラリによっては、ハンドルを Arc/Rc で包んだ状態のものが提供されていることがある。これらはクローンでき、それらは同じものを参照している。

ライフタイムと排他制御

ライフタイムと排他制御: 概要

- ライフタイムはスタック領域のためと思われがちだが、静的な排他制御をするというもう一つの役割がある。
- 参照の排他性は時間分割か空間分割によって保障される。ライフタイムは時間分割を正当化するという重要な役割がある。
- 時間分割が強制されることにより、ある種の曖昧なコードを防ぐことができる。これは一方で、はまりやすいコンパイルエラーの一種でもある。

ライフタイムの存在目的

- スタック領域を安全に使うため
- 静的な排他制御のため

スタック領域の安全性

- 安全でない例（コンパイルエラー）

```
fn foo() -> &i32 {  
    let x = 42;  
    &x  
}
```


スタック領域の安全性

- 安全でない例 (コンパイルエラー)

```
fn foo() -> &i32 {  
    let x = 42;  
    &x  
}
```

foo 用のスタックフレーム上に
42が置かれる

この参照は foo 内では有効ではない！
関数外に返せてしまうと問題になる

スタック領域の安全性

- 安全でない例（コンパイルエラー）

```
fn foo() -> &i32 {
```

多くの言語では、ユーザーが扱えるポインタはGC管理されており、常にヒープを指すので、このような管理は要らない。そのかわりヒープ管理とGCのコストが追加される。

GoやML Kitなどの一部の言語には、安全だとわかっているときだけ自動的にスタックに確保するような仕組みがある。

Rustがそのような自動化をしない理由はいくつか考えられる。たとえば、RustはCと同様、ヒープへの依存を切り離せるように設計されている。そのため、ヒープがデフォルトになる仕組みが言語に入ることはあまり考えられない。

静的な排他制御のためのライフタイム

- Q. `src` と `dst` が同じだったら何が起こる？

```
fn append(src: &Vec<u32>, dst: &mut Vec<u32>) {  
    for &x in src {  
        dst.push(x);  
    }  
}
```

静的な排他制御のためのライフタイム

- Q. src と dst が同じだったら何が起こる？
- A. src と dst が**同じになることはない**

```
fn append(src: &Vec<u32>, dst: &mut Vec<u32>) {  
    for &x in src {  
        dst.push(x);  
    }  
}
```

静的な排他制御のためのライフタイム

- Q. src と dst が同じだったら何が起こる？
- A. src と dst が同じになることはない

コンパイラ: コーナーケースを考えなくてよいので嬉しい！

プログラマ: コーナーケースを考えなくてよいので嬉しい！

静的な排他制御のためのライフタイム

- 呼んでみよう！

```
fn append(src: &Vec<u32>, dst: &mut Vec<u32>) {  
    for &x in src {  
        dst.push(x);  
    }  
}  
  
fn main() {  
    let mut v = vec![1, 2, 3];  
    append(&v, &mut v);  
    println!("{:?}", v);  
}
```

復習: 排他制御のルール

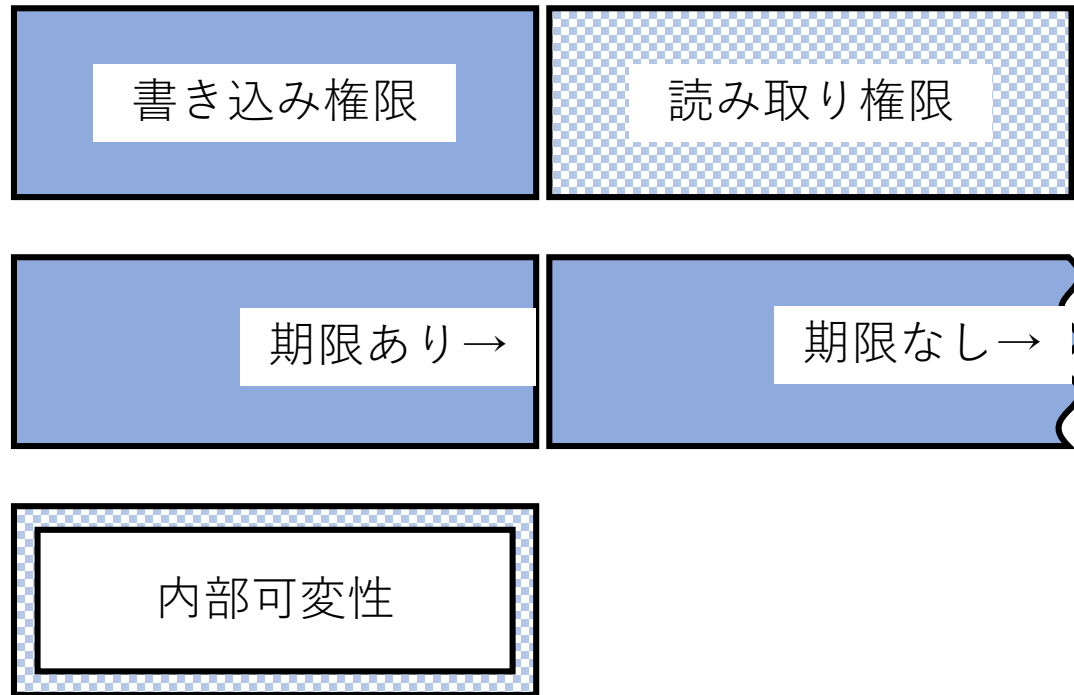
- 同じ時刻に、同じ領域に対して、
 - 書き込み・書き込み→NG
 - 書き込み・読み取り→NG
 - 読み取り・読み取り→OK

図でわかる Rust の排他制御

まずは何となく眺めてください

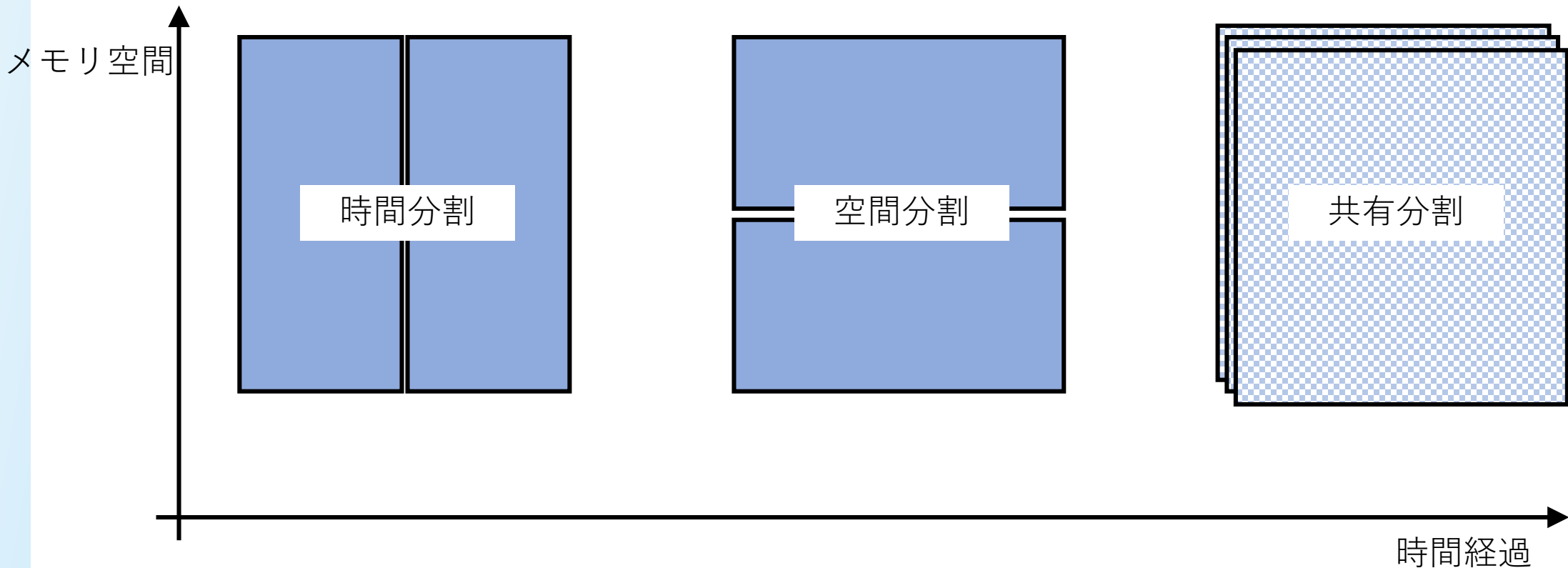
図でわかる Rust の排他制御

- 凡例



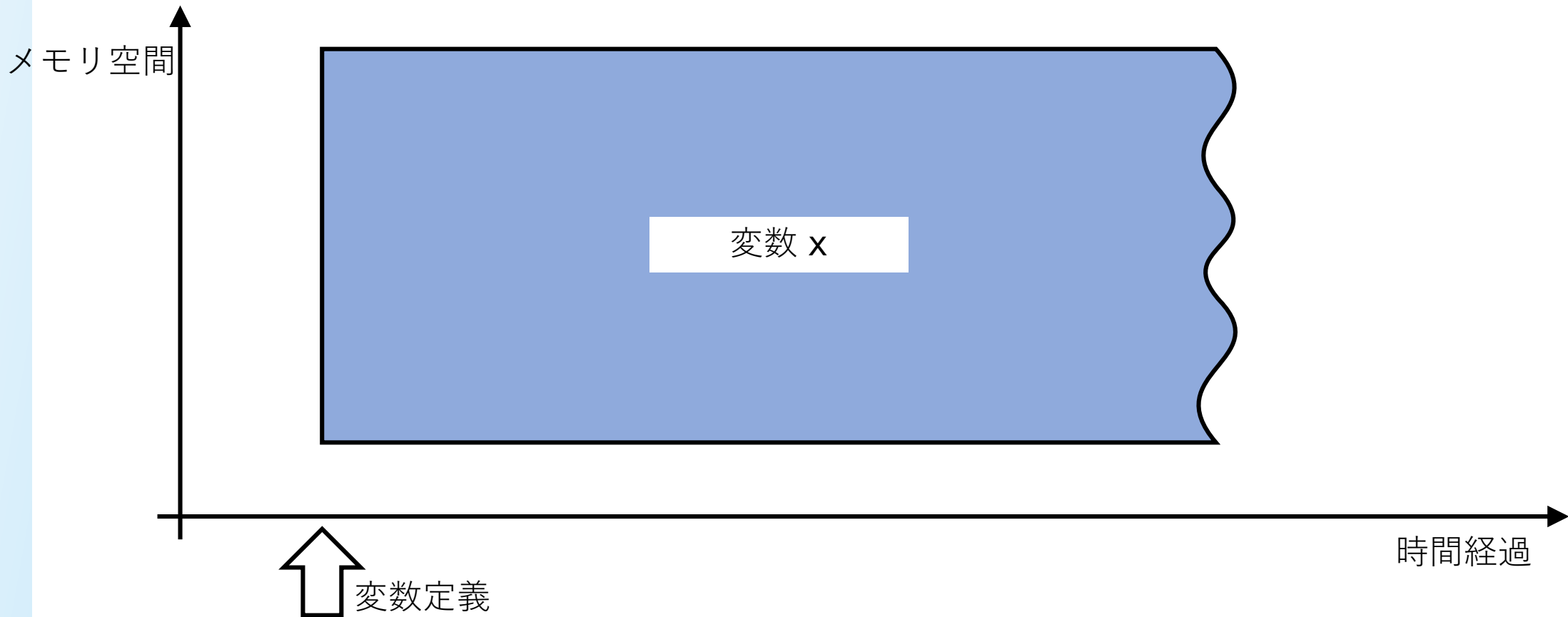
図でわかるRustの排他制御

- 所有権・借用は3つの方法で分割できる



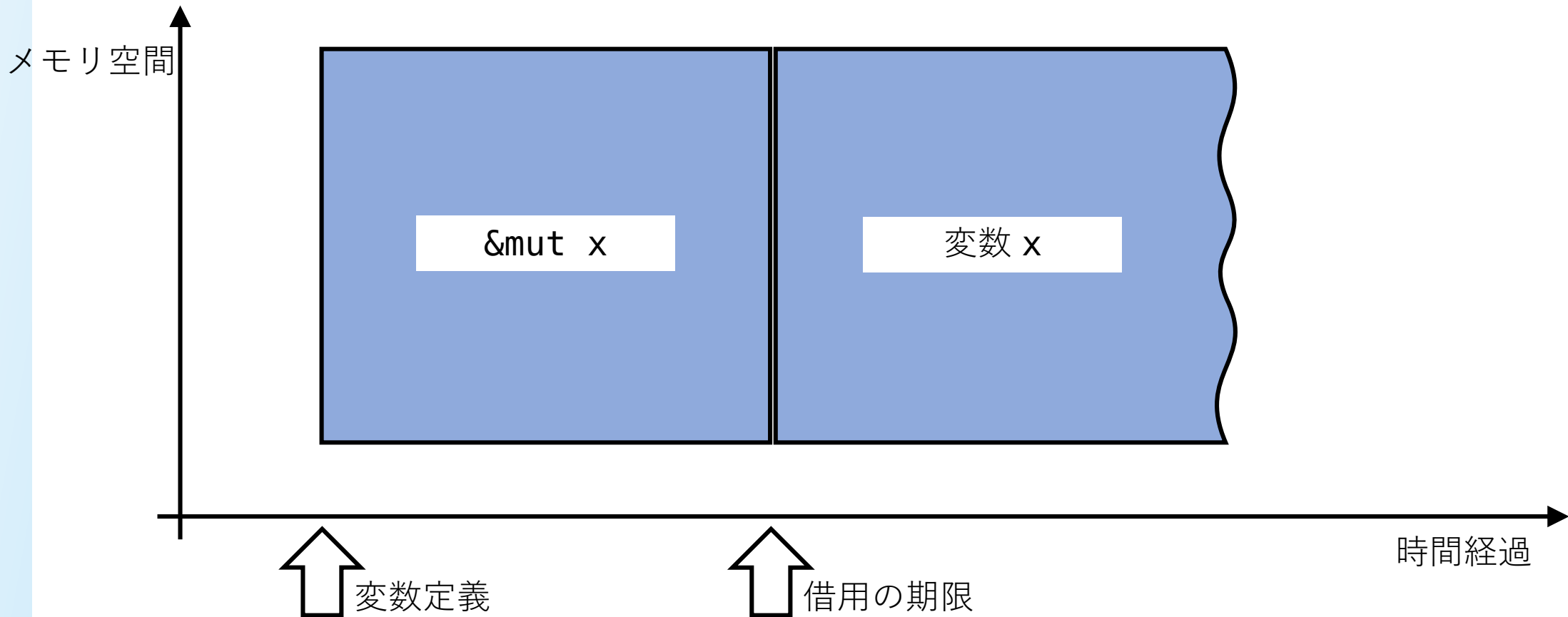
図でわかる Rust の排他制御

- 変数を宣言した瞬間



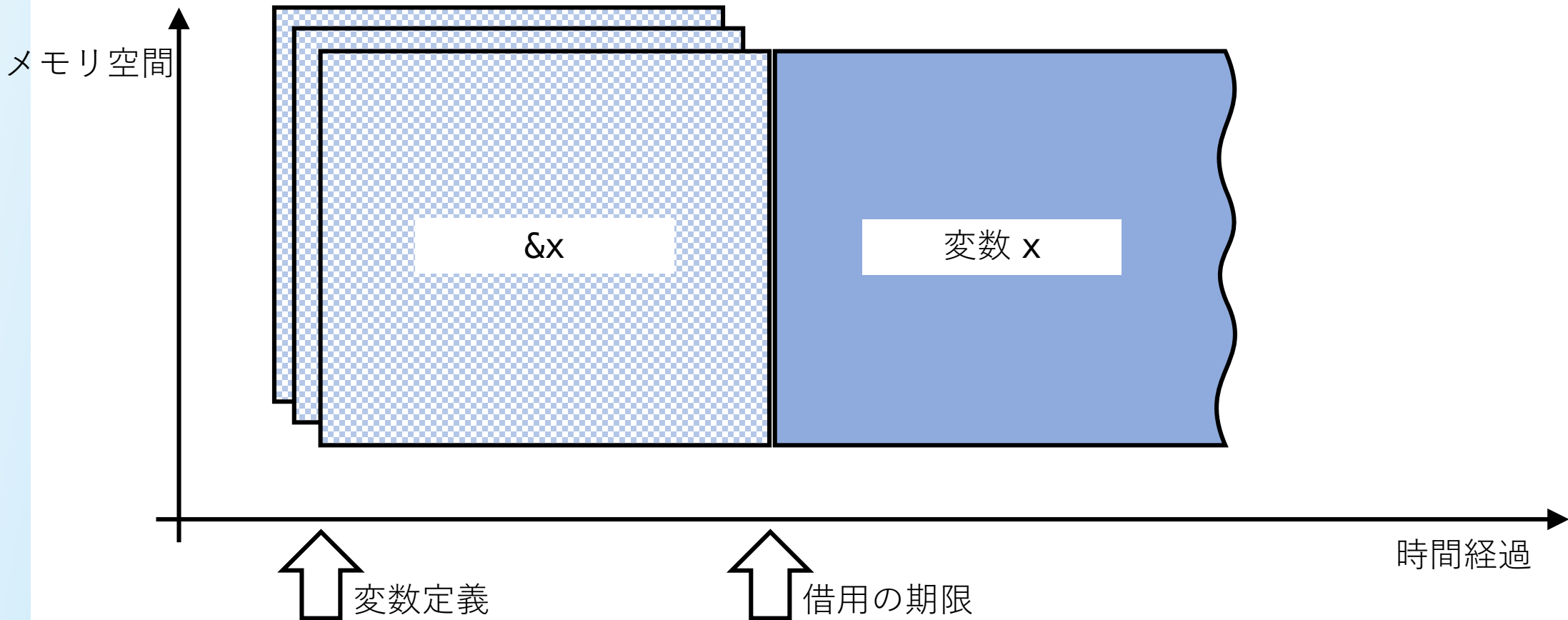
図でわかる Rust の排他制御

- `&mut` 参照の取得 = **時間分割**



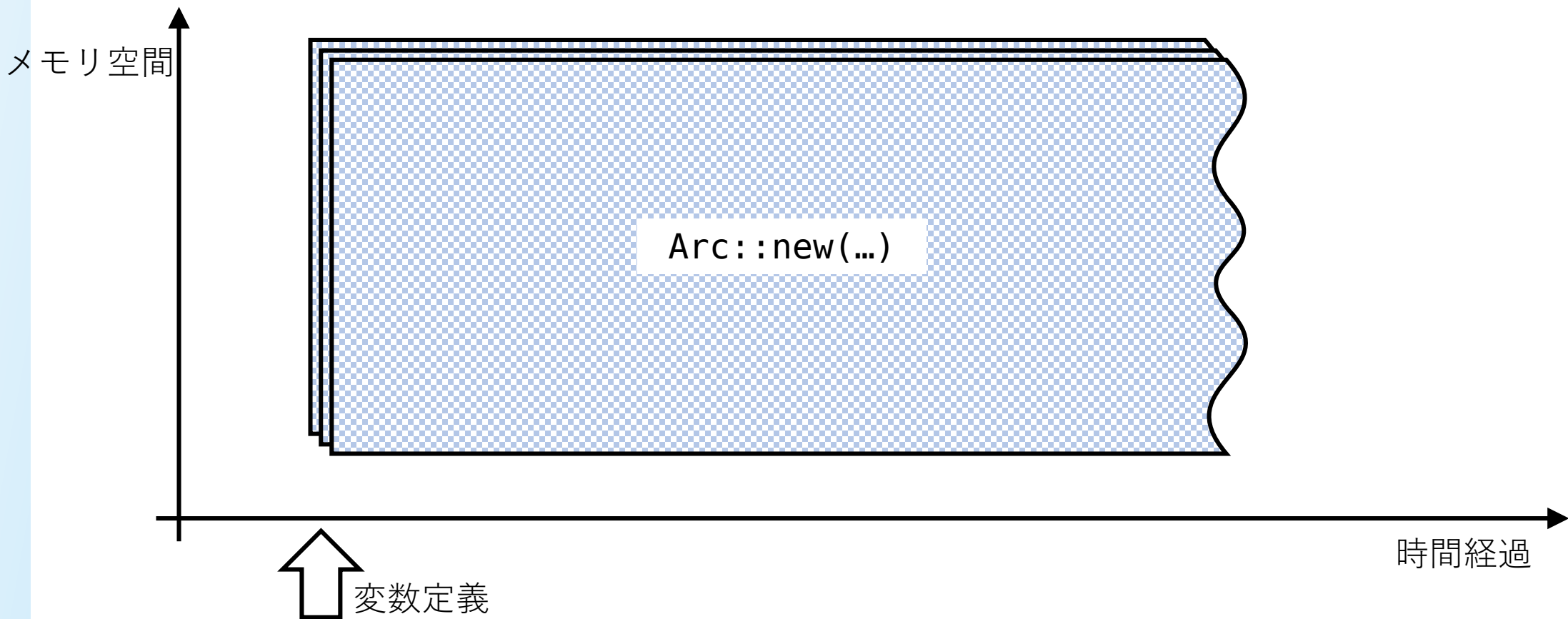
図でわかる Rust の排他制御

- **&** 参照の取得 = **共有分割** (+時間分割)



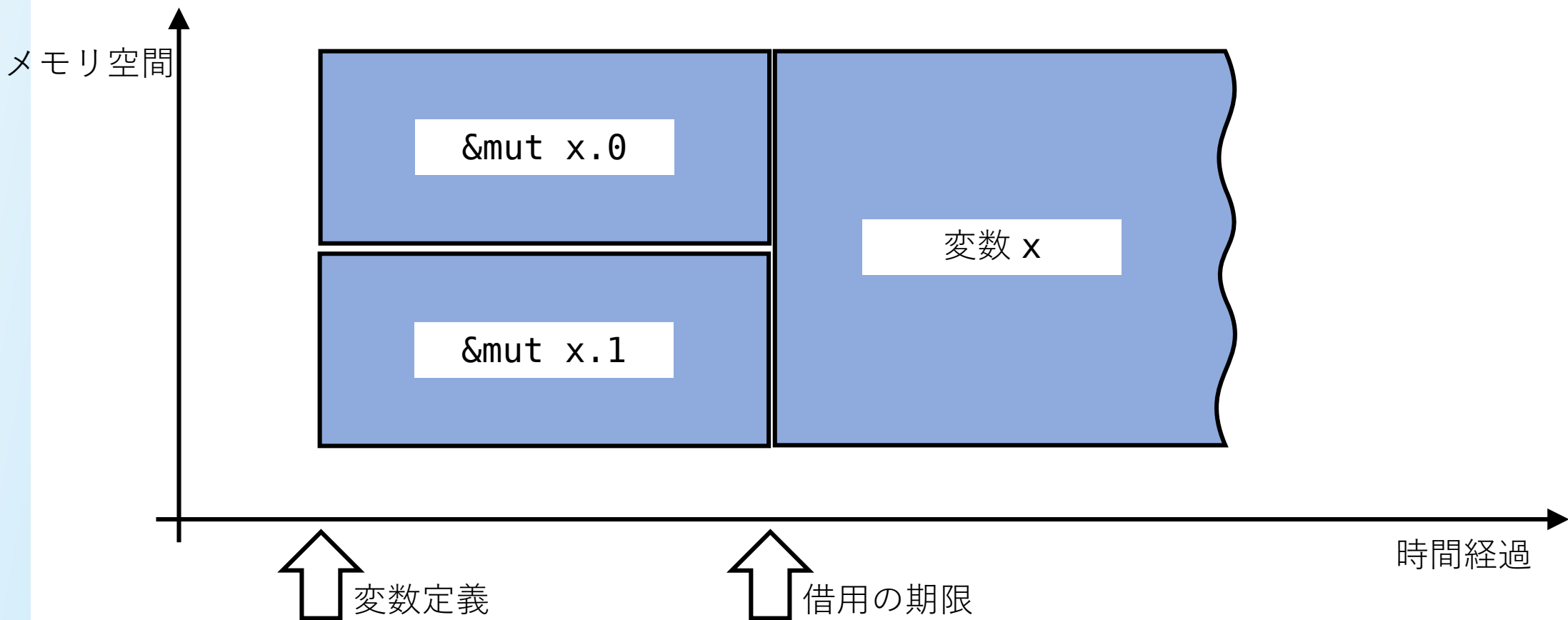
図でわかる Rust の排他制御

- Arc = 共有分割



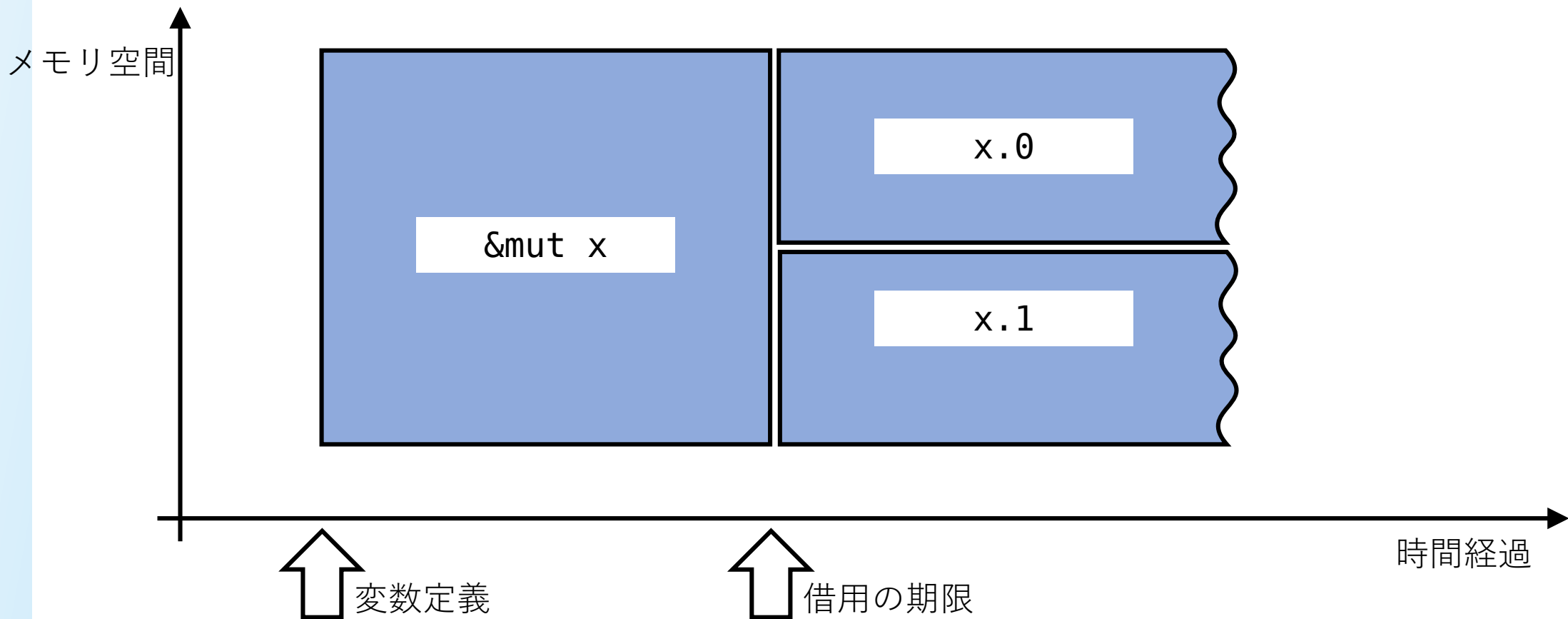
図でわかる Rust の排他制御

- `&mut` 参照の**空間分割** (構造体の分割借用、`split_at_mut`)



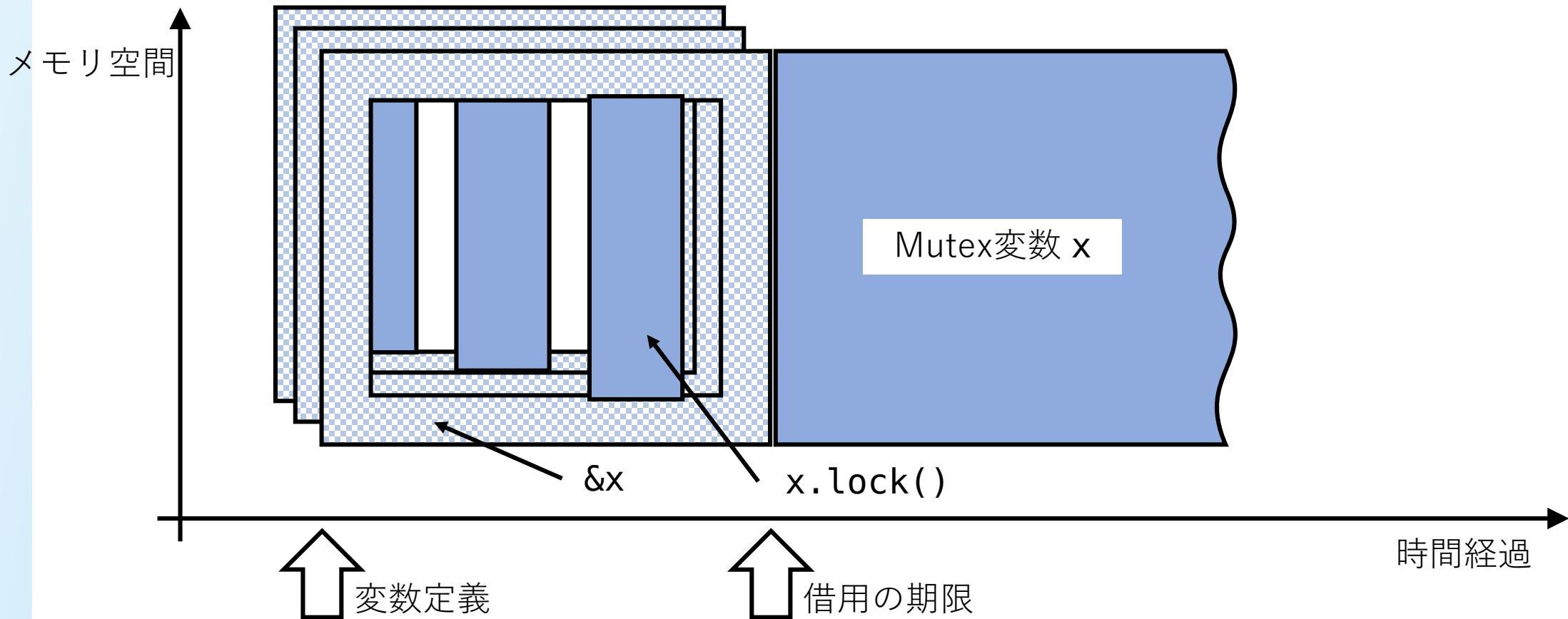
図でわかる Rust の排他制御

- 所有権の**空間分割**（構造体の分割、`drain`）



図でわかるRustの排他制御

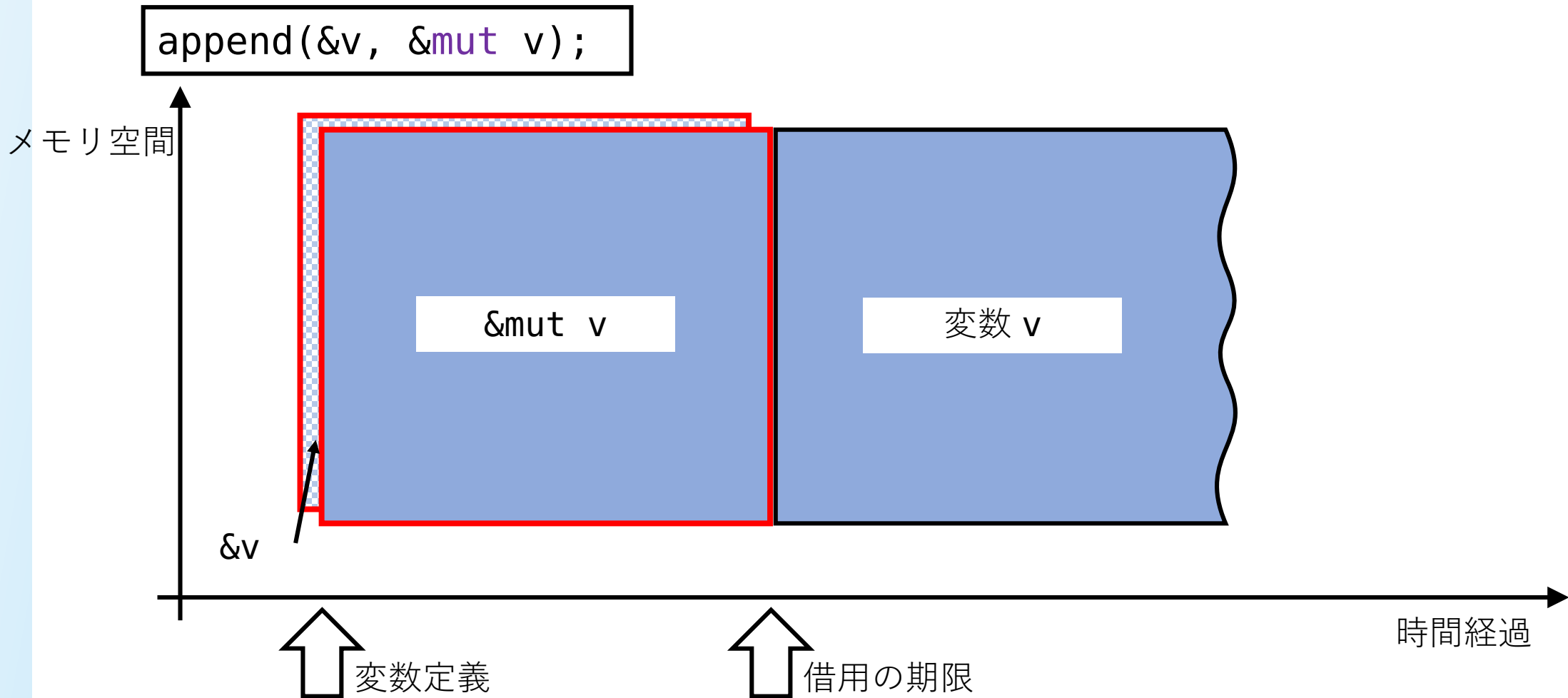
- Mutex: 枠を共有し、中身を動的に排他制御する



図でわかる Rust の排他制御

何となく、雰囲気だけでもわかりましたか？
では具体例を見てみましょう

例: セルフappendが呼べない理由



例: ループ内での自己改変

```
let mut v = vec![8, 9, 10];  
  
for &x in &v {  
    if x > 0 {  
        v.push(x / 2);  
    }  
}
```

例: ループ内での自己改変

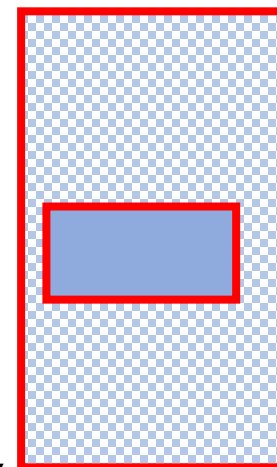
```
let mut v = vec![8, 9, 10];
```

```
for &x in &v {  
    if x > 0 {  
        v.push(x / 2);  
    }  
}
```

ループの間 v を借りている

書き込みがイテレーターの読み取りと競合する

時間経過



例: ループ内での自己改変

- そもそも、このコードは期待する実行結果が不明瞭
- パターン1: [8, 9, 10, 4, 4, 5]
(伸びた分のループは実行されない)
- パターン2: [8, 9, 10, 4, 4, 5, 2, 2, 2, 1, 1, 1, 0, 0, 0]
(伸びた分のループも実行される)

例: ループ内での自己改変

- `len` の更新を無視してループしたい場合 (1)

```
let mut v = vec![8, 9, 10];  
  
for x in v.clone() {  
    if x > 0 {  
        v.push(x / 2);  
    }  
}
```

例: ループ内での自己改変

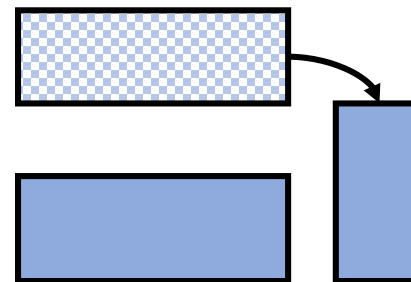
- len の更新を無視してループしたい場合 (1)

```
let mut v = ve
```

v をループ開始前に複製してしまう。

```
for x in v.clone() {  
    if x > 0 {  
        v.push(x / 2);  
    }  
}
```

時間経過



例: ループ内での自己改変

- `len` の更新を無視してループしたい場合 (2)

```
let mut v = vec![8, 9, 10];  
  
for i in 0..v.len() {  
    let x = v[i];  
    if x > 0 {  
        v.push(x / 2);  
    }  
}
```

例: ループ内での自己改変

- `len` の更新を無視してループしたい場合 (2)

```
let mut v = ve
```

v をループ開始時に一瞬だけ借りている。
一度覚えた長さをずっと使う。

```
for i in 0..v.len() {  
    let x = v[i];  
    if x > 0 {  
        v.push(x / 2);  
    }  
}
```



例: ループ内での自己改変

- `len` の更新を反映してループしたい場合

```
let mut v = vec![8, 9, 10];

let mut i = 0;
while i < v.len() {
    let x = v[i];
    if x > 0 {
        v.push(x / 2);
    }
    i += 1;
}
```

例: ループ内での自己改変

- `len` の更新を反映してループしたい場合

```
let mut v = vec![8, 9, 10];
```

```
let mut i = 0;
```

```
while i < v.len() {
```

```
    let x = v[i];
```

```
    if x > 0 {
```

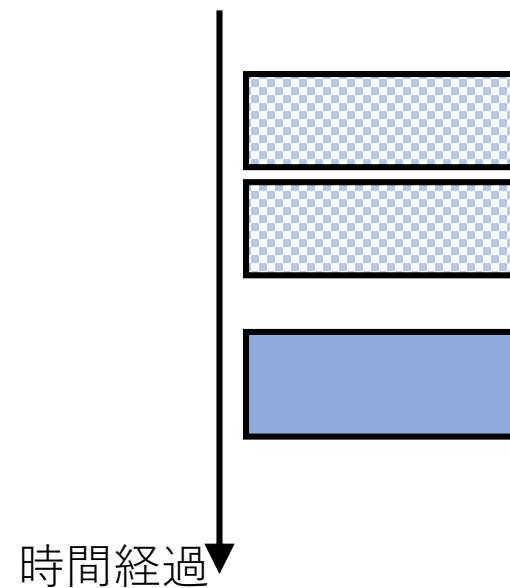
```
        v.push(x / 2);
```

```
    }
```

```
    i += 1;
```

```
}
```

v の長さを毎回取りに行く



他の言語では……

- できるだけ同じ構造のコードを書いて実験してみた
- [Ruby](#): [8, 9, 10, 4, 4, 5, 2, 2, 2, 1, 1, 1, 0, 0, 0]
- [Python](#): [8, 9, 10, 4, 4, 5, 2, 2, 2, 1, 1, 1, 0, 0, 0]
- [Java](#): java.util.ConcurrentModificationException
- [C++](#): [8, 9, 10, 4, 5] (おそらく未定義動作)
- [Go](#): [8, 9, 10, 4, 4, 5]
- JavaとGoはこのケースを意識的にハンドリングしてる感じがある

継承可変性と排他制御

継承可変性と排他制御: 概要

- Rustのイミュータビリティーは継承される。これは排他制御の原理から理解することができる。
- イミュータビリティーが継承されることで、ネストしたデータ構造の意図しない書き換えを防止できる。

継承可変性と内部可変性

継承可変性

(**inherited** mutability)

フィールドや参照先に
可変性が継承される

Rustではこっちが原則

内部可変性

(**interior** mutability)

不変な構造体の内側に
可変な値をとれる

**Rustでは内部可変性を
オプトインする**

例: グラフオブジェクト

- グラフをインメモリの隣接リストで保持するオブジェクトを考える。
- AddVertex() -> Integer
- AddEdge(a: Integer, b: Integer)
- GetGraph() -> Array<Array<Integer>>

Rubyの場合

```
class Graph
  attr_reader :graph
  def initialize; @graph = []; end
  def add_vertex; @graph << []; @graph.len - 1; end
  def add_edge(a, b); @graph[a] << b; @graph[b] << a; end
end
```

Pythonの場合

```
class Graph(object):  
    def __init__(self): self._graph = []  
    def add_vertex(self):  
        self._graph.append([])  
        return len(self._graph) - 1  
    def add_edge(self, a, b):  
        self._graph[a].append(b)  
        self._graph[b].append(a)  
    @property  
    def graph(self): return self._graph
```

例: グラフオブジェクト

- グラフをインメモリの隣接リストで保持するオブジェクトを考える。
- AddVertex() -> Integer
- AddEdge(a: Integer, b: Integer)
- GetGraph() -> Array<Array<Integer>>
- **追加の要件:** GetGraphはイミュータブルなビューを返す

RubyやPythonでは……

- 方法1: **ディープコピー**する
- 方法2: **ディープコピーをキャッシュ**する
- 方法3: 専用のビュークラスを作る（再帰的にビューを返す）

Rustの場合

```
struct Graph { graph: Vec<Vec<usize>>, }

impl Graph {
    pub fn new() -> Self { Self { graph: Vec::new() } }
    pub fn add_vertex(&mut self) { self.graph.push(Vec::new()); }
    pub fn add_edge(&mut self, a: usize, b: usize) {
        self.graph[a].push(b);
        self.graph[b].push(a);
    }
    pub fn graph(&self) -> &[Vec<usize>] {
        &self.graph
    }
}
```

イミュータブルなスライス

Rustの場合

```
struct Graph { graph: Vec<Vec<usize>>, }

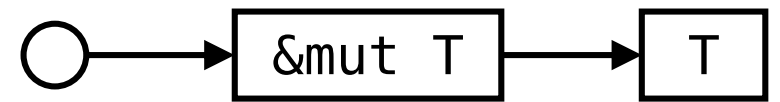
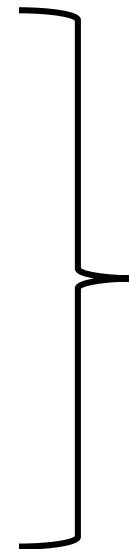
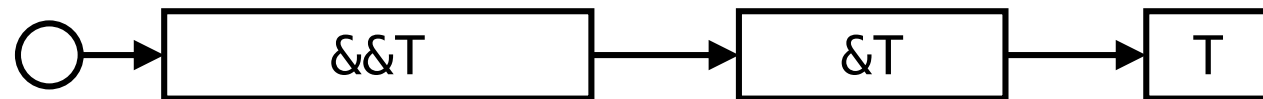
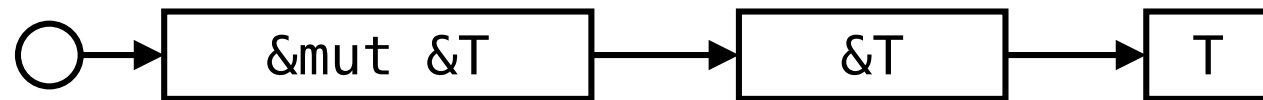
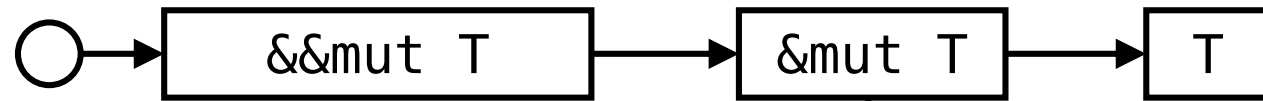
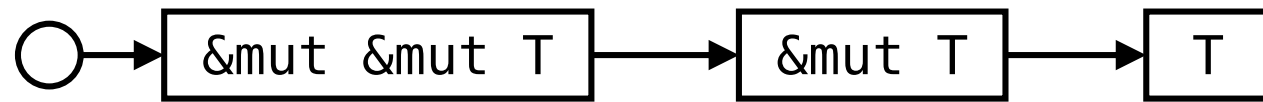
impl Graph {
    pub fn new() -> Self { Self { graph: Vec::new() } }
    pub fn add_vertex(&mut self) { self.graph.push(Vec::new()); }
    pub fn add_edge(&mut self, a: usize, b: usize) {
        self.graph[a].push(b);
        self.graph[b].push(a);
    }
    pub fn graph(&self) -> &[Vec<usize>] {
        &self.graph
    }
}
```

内側も自動的にイミュータブルになる（継承可変性）

再掲: 排他制御のルールと参照

- 同じ時刻に、同じ領域に対して、
 - 書き込み・書き込み→NG
 - 書き込み・読み取り→NG
 - 読み取り・読み取り→OK
- Rustの参照:
 - `&T` は**共有参照** → 共有されているので(原則)読み取り専用
 - `&mut T` は**排他参照** → 排他的なので読み書き可能

参照の参照



内部可変性と排他制御

内部可変性と排他制御: 概要

- 継承可変性では不十分な場合には、共有書き換えの仕組みをオプトインすることができる。これを内部可変性という。
- 内部可変性は、動的な排他制御として一貫して理解することができる。

継承可変性と内部可変性

継承可変性

(**inherited** mutability)

フィールドや参照先に
可変性が継承される

Rustではこっちが原則

内部可変性

(**interior** mutability)

不変な構造体の内側に
可変な値をとれる

**Rustでは内部可変性を
オプトインする**

例: 一意IDを振る君

- UUIDやsnowflakeのような一意IDを振るプロセス内サービスを考える

例: 一意IDを振る君

- だいたいこんな感じのラッパー型を作る

```
[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct Id(u32);

impl Id {
    pub fn id(&self) -> u32 {
        self.0
    }
}
```

例: 一意IDを振る君

- 一意IDを振る君の構造体を定義する

```
#[derive(Debug)]
pub struct GenId {
    next: u32,
}

impl GenId {
    pub fn new() -> Self {
        Self {
            next: 0,
        }
    }
}
```

例: 一意IDを振る君

- 一意IDを振る君の共有方法はいくつかある
- 親玉が生成した `&GenId` を持ち回す
- `Arc` に包んで `Arc<GenId>` として複製していく
- `lazy_static!` でグローバル変数として共有する
- いずれにしても、 `GenId` は**共有される**というのがミソ

例: 一意IDを振る君

- 共有していても、next は書き換えたい

```
#[derive(Debug)]
pub struct GenId {
    next: u32,
}

impl GenId {
    pub fn new() -> Self {
        Self {
            next: 0,
        }
    }
}
```

共有中は書き換えられない

例: 一意IDを振る君

- **Mutex** などで内部可変性を導入する

```
#[derive(Debug)]
pub struct GenId {
    next: Mutex<u32>,
}

impl GenId {
    pub fn new() -> Self {
        Self {
            next: Mutex::new(0),
        }
    }
}
```

共有中は書き換えられない

Rustの **Mutex** は「ミューテックス本体」と「それによって守られるデータ」の対になっている。
Mutex の中に入れば、ロックが必要になる。
Mutex の外に出せば、共有書き込みはできない。
どちらにしても、危険な動作はできない。

例: 一意IDを振る君

- 使うときはロックを取得する

```
impl GenId {  
    pub fn gen(&self) -> Id {  
        let mut next = self.next.lock().unwrap();  
        let id = Id(*next);  
        *next += 1;  
        id  
    }  
}
```

ロックを取得

ここで next がドロップされる。
ロックも自動的に解放される。

例: 一意IDを振る君

```
impl GenId {  
    pub fn gen(&self) -> Id {  
        let mut next = self.next.lock().unwrap();  
        let id = Id(*next);  
        *next += 1;  
        id  
    }  
}
```

ロック中は `&mut` 参照のように扱うことができる

例: 一意IDを振る君

- ロックの粒度

```
impl GenId {  
    pub fn gen(&self) -> Id {  
        let mut next = self.next.lock().unwrap();  
        let id = Id(*next);  
        drop(next);  
  
        let mut next = self.next.lock().unwrap();  
        *next += 1;  
        id  
    }  
}
```

わざわざこう書く人はたぶんいないが……
こうすると一旦ロックを解放するので不整合が起きる

例: 一意IDを振る君

- ロックの粒度

```
impl GenId {  
    pub fn gen(&self) -> Id {  
        let mut next = self.next.lock().unwrap();  
        let id = Id(*next);  
        drop(next);  
  
        let mut next = self.next.lock().unwrap();  
        *next += 1;  
        id  
    }  
}
```

逆に言えば、ロックを握っている間はちゃんと専有しているので、知らない変更が紛れることはない

内部可変性とスレッド安全性

内部可変性とスレッド安全性: 概要

- 継承可変性と異なり、内部可変性は実装によってはスレッド安全ではない。
- Rustでは、スレッド安全な実装とシングルスレッド実装を選択できる。
- シングルスレッド実装を選んでも、危険なコードは未然に防止される。

継承可変性と内部可変性

継承可変性

(**inherited** mutability)

フィールドや参照先に
可変性が継承される

Rustではこっちが原則

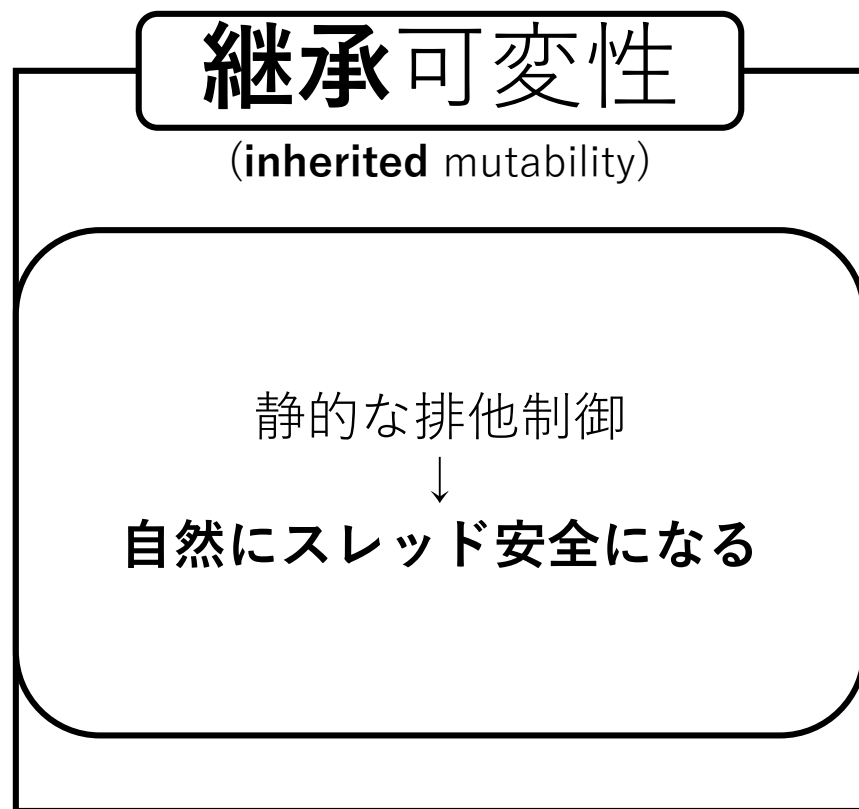
内部可変性

(**interior** mutability)

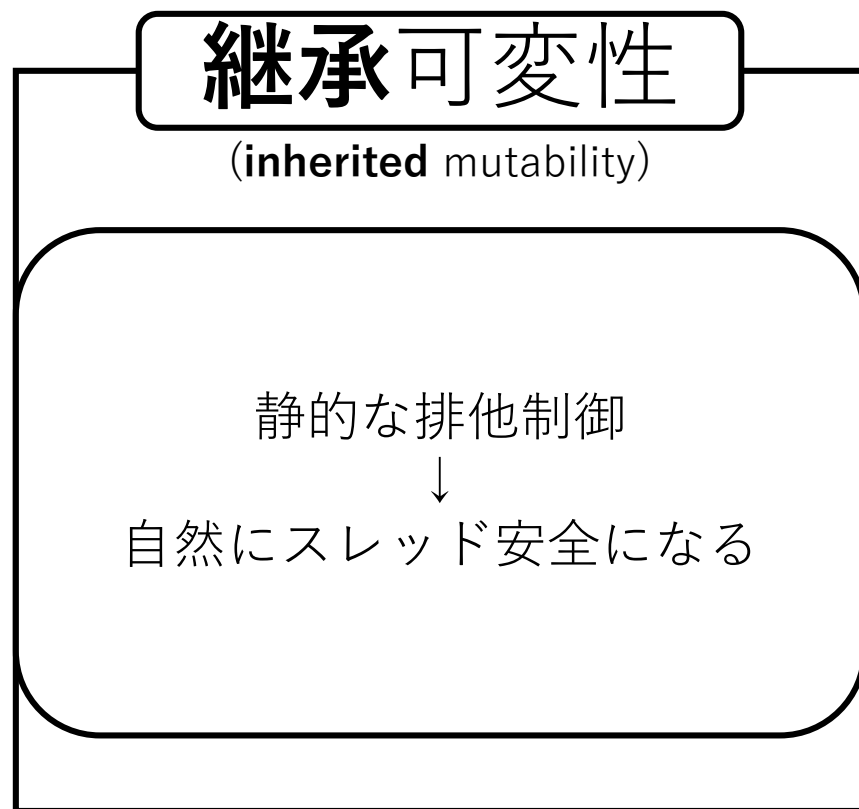
不変な構造体の内側に
可変な値をとれる

**Rustでは内部可変性を
オプトインする**

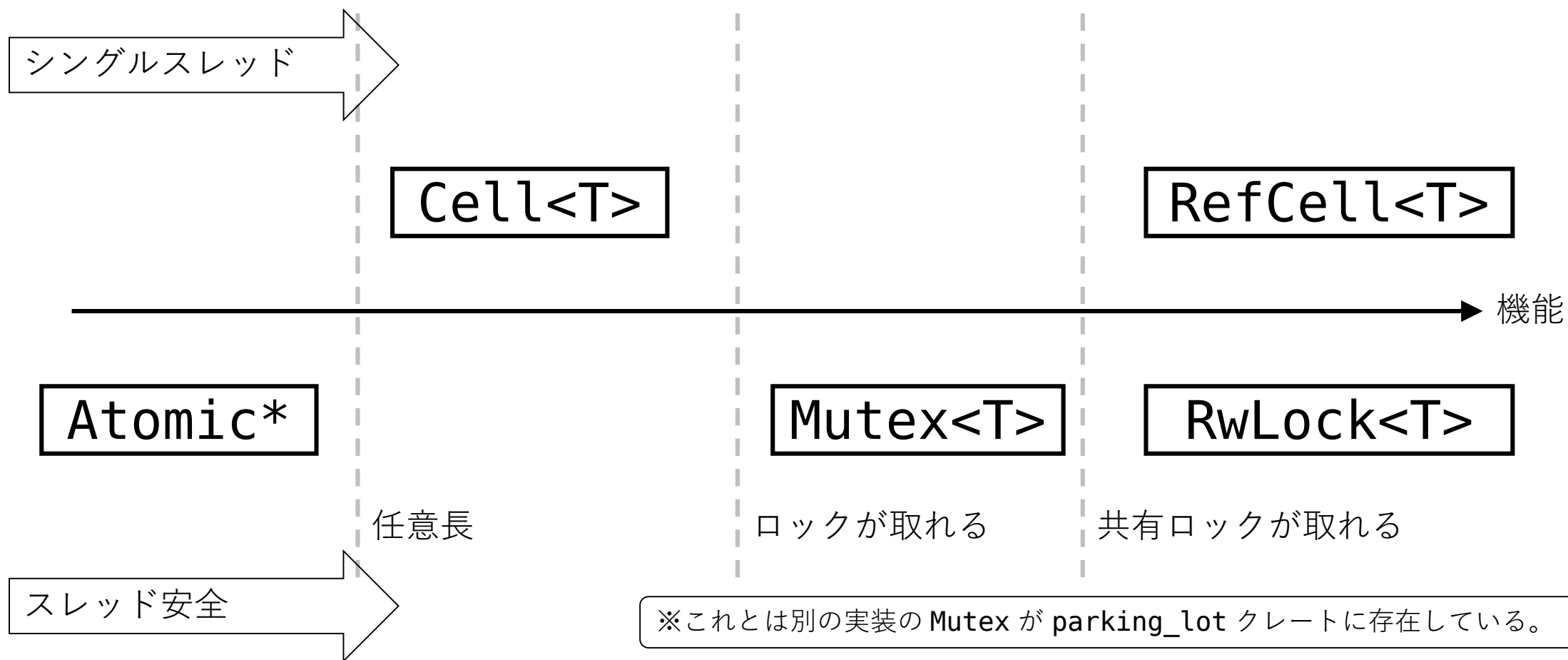
継承可変性とスレッド安全性



継承可変性とスレッド安全性



内部可変性の亜種たち



Arc と Rc

- Arc と Rc は**内部可変性を作るのには使えない**。
- しかし、参照カウンタのために内部可変性を使っているため、スレッドセーフな Arc とシングルスレッド用の Rc がある。

※単一のロックで参照カウンタと参照先の両方をロックするような実装も考えられる。もちろん、長所と短所がある。
Rustでは粒度を細かくするほうを選択したようだ。

スレッド安全性を守る

- 例: rayon を使って並列に何かを計算してみる

```
extern crate rayon;

use rayon::prelude::*;

fn main() {
    let sum = (0u64..1000).into_par_iter().map(|n| n * n).sum::<u64>();
    println!("sum = {}", sum);
}
```

スレッド安全性を守る

- 例: パフォーマンススカウンタ的なことをやりたくなった

```
fn main() {  
    let mut counter = 0;  
    let sum = (0u64..1000)  
        .into_par_iter()  
        .map(|n| {  
            counter += 1;  
            n * n  
        })  
        .sum::<u64>();  
    println!("sum = {}", sum);  
    println!("counter = {}", counter);  
}
```

スレッド安全性を守る

- 例: パフォーマンススカウンタ的なことをやりたくなった

```
fn main() {  
    let mut counter = 0;  
    let sum = (0u64..1000)  
        .into_par_iter()  
        .map(|n| {  
            counter += 1;  
            n * n  
        })  
        .sum::<u64>();  
    println!("sum = {}", sum);  
    println!("counter = {}", counter);  
}
```

普通の(直列)イテレーターならこれでコンパイルできる。
(順番に実行されるから、環境をmutキャプチャー可能)

スレッド安全性を守る

- 例: パフォーマンススカウンタ的なことをやりたくなった

```
fn main() {  
    let mut counter = 0;  
    let sum = (0u64..1000)  
        .into_par_iter()  
        .map(|n| {  
            counter += 1;  
            n * n  
        })  
        .sum::<u64>();  
    println!("sum = {}", sum);  
    println!("counter = {}", counter);  
}
```

並列イテレーターではエラーになってしまう。
(counter への参照を各所で共有するため)

スレッド安全性を守る

- 例: 内部可変性を導入する

```
fn main() {  
    let counter = Cell::new(0);  
    let sum = (0u64..1000)  
        .into_par_iter()  
        .map(|n| {  
            counter.set(counter.get() + 1);  
            n * n  
        })  
        .sum::<u64>();  
    println!("sum = {}", sum);  
    println!("counter = {}", counter.get());  
}
```

スレッド安全ではない！
コンパイルエラーになる。

スレッド安全性を守る

- 例: スレッド安全な内部可変性を導入する

```
fn main() {  
    let counter = AtomicUsize::new(0);  
    let sum = (0u64..1000)  
        .into_par_iter()  
        .map(|n| {  
            counter.fetch_add(1, atomic::Ordering::SeqCst);  
            n * n  
        })  
        .sum::<u64>();  
    println!("sum = {}", sum);  
    println!("counter = {}", counter.load(atomic::Ordering::SeqCst));  
}
```

スレッド安全！

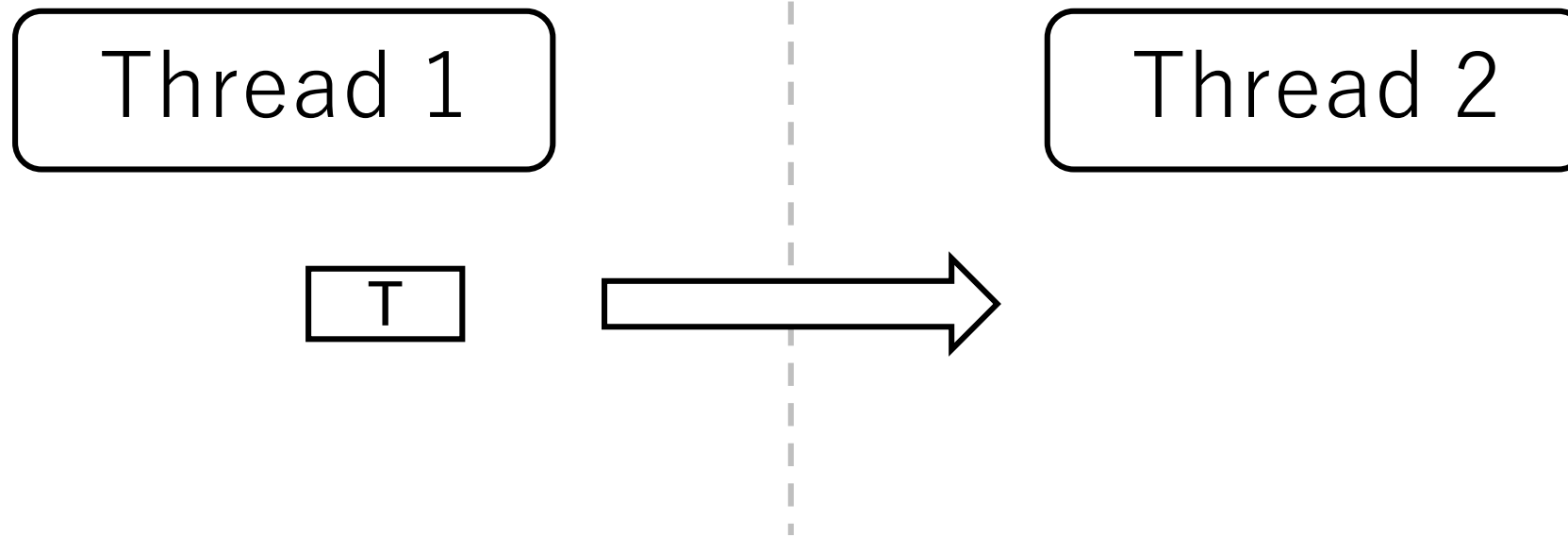
※本当は AtomicU32 を使いたいが、これは移植性に関する仕様が定まっていないためnightlyでしか使えない。

Send と Sync

- スレッド安全性は **Send** と **Sync** というマーカートレイトによって再帰的に検査される。

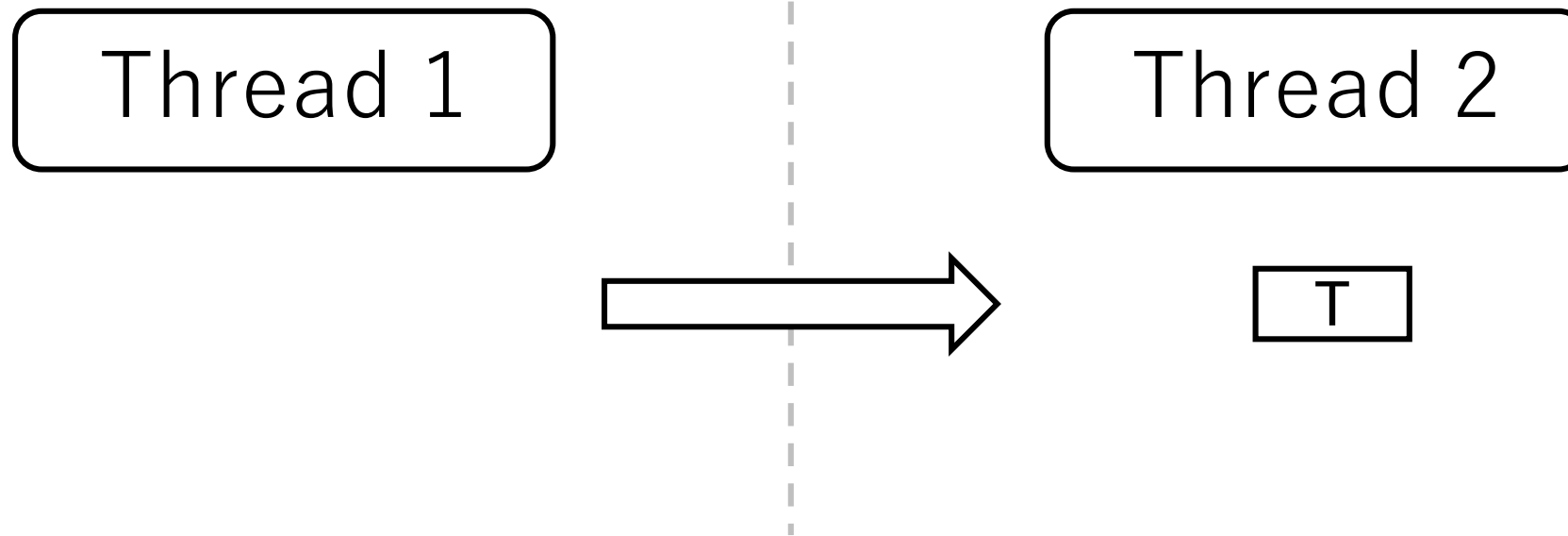
Send と Sync

- Send: 全体を別スレッドに送ることができる。



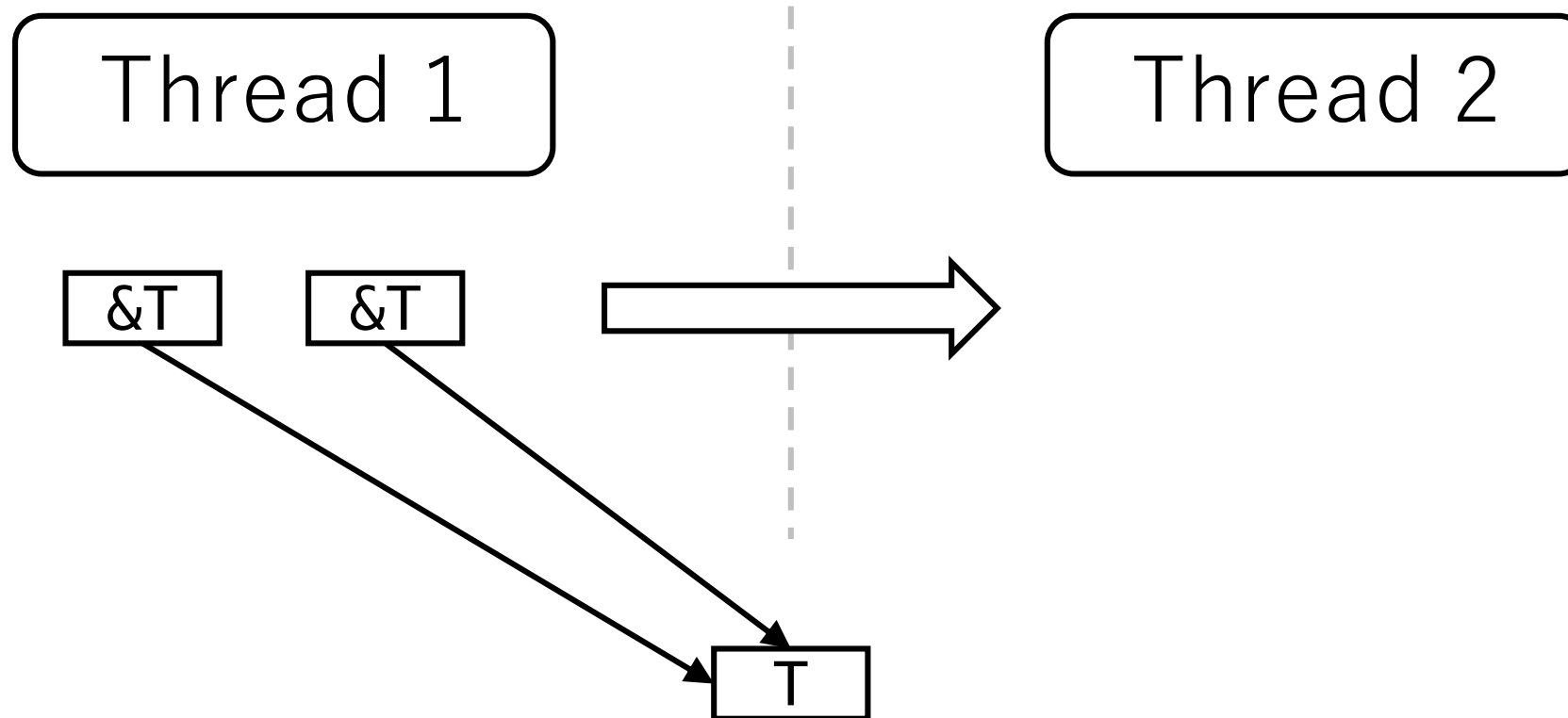
Send と Sync

- Send: 全体を別スレッドに送ることができる。



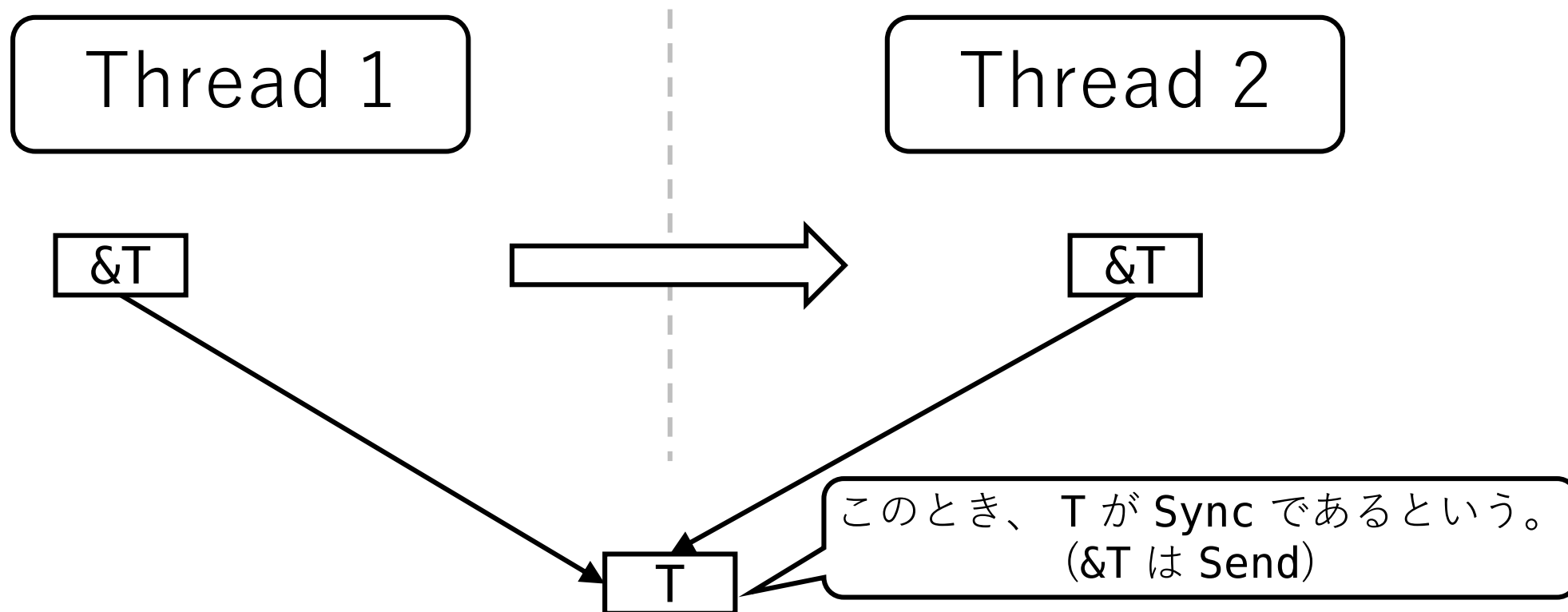
Send と Sync

- Sync: 別スレッドと共有することができる。



Send と Sync

- Sync: 別スレッドと共有することができる。



Mutex の特殊能力

- Mutex は Send を Send + Sync に格上げできる。
- RwLock にこの能力はない

```
impl<T: ?Sized + Send> Send for Mutex<T>
```

```
impl<T: ?Sized + Send> Sync for Mutex<T>
```

```
impl<T: ?Sized + Send> Send for RwLock<T>
```

```
impl<T: ?Sized + Send + Sync> Sync for RwLock<T>
```

※ RwLock ではなく Mutex を使う他の理由として、書き込み飢餓状態を避けたいというケースも考えられる。

スレッド保証をどう選ぶか

- アプリケーションコードでは、シングルスレッドで始めて、困ってからスレッド安全な実装に切り替えてもよい。
- ライブラリでは、ユーザーが **Send** を必要としているかどうかちゃんと考えてほしいかも……
 - ひとたび **Rc** を使ってしまうと、その値はどう頑張っても他のスレッドに逃げられない。

クロージャが難しくて……

苦勞じゃ

クロージャが難しくて……: 概要

- クロージャは関数と違ってキャプチャーをするので、キャプチャーと所有権の関わりでハマる可能性がある。
- 実際のところ、使うときは **move** をつけるかつけないかくらいしか判断するところはない。たいてい、クロージャが長生きする必要があるときに **move** をつける。
- それに加えて、クロージャのインターフェースを設計するときには、**FnOnce/FnMut/Fn** を選ぶ必要がある。
- 外側の制御フローへの干渉はできない。

クロージャの構文

- パイプ2本を前置すると、クロージャになる。

```
|x| x * a
```

- 引数の型は省略できる。明示するときにはこんな感じ。

```
|x: i32, y: i32| x * a + y
```

- 戻り値の型を明示するときには、`{}` が必要。

```
|| -> i32 { 42 }
```

- `move` を前置できる(後述)。これはorではない。

```
move || x
```

Rustにおけるクロージャと関数

- どちらも、定義ごとに異なる型を持つ。
- どちらも、 **Fn*** トraitを実装し、 **f()** 構文で呼び出せる。
- 関数は多相だが、クロージャは多相ではない。
- クロージャは再帰できない。
- 関数は外のローカル変数を参照できない。クロージャは外のローカル変数を参照できる(**キャプチャー**)。

※C/C++の関数は定義ごとに異なる型を持つわけではない。
※C++は多相クロージャを持つし、クロージャによる再帰もできる。

キャプチャーの仕組み

- 例: 合計を仰々しく計算する

```
fn main() {  
    let mut sum = 0;  
    (0..10)  
        .map(|x| {  
            sum += x;  
        })  
        .collect::<()>();  
    println!("{}", sum);  
}
```

※例示のために、わざとクロージャを使っている。
普通は **for** ループで自分で足すか、**sum** メソッドを使う。

キャプチャーの仕組み

- 例: 合計を仰々しく計算する

```
fn main() {  
    let mut sum = 0;  
    (0..10)  
        .map(|x| {  
            sum += x;  
        })  
        .collect::<()>();  
    println!("{}", sum);  
}
```

このクロージャは sum に書き込みアクセスしているので

```
struct closure {  
    sum: &mut i32,  
}
```

```
*self.sum += x;
```

※例示のために、わざとクロージャを使っている。
普通は for ループで自分で足すか、sum メソッドを使う。

キャプチャーの仕組み

- 例: 合計を仰々しく計算する

```
fn main() {  
    let mut sum = 0;  
    (0..10)  
        .map(|x| {  
            sum += x;  
        })  
        .collect::<()>();  
    println!("{}", sum);  
}
```

このクロージャは sum に書き込みアクセスしているので

```
struct closure {  
    sum: &mut i32,  
}
```

```
*self.sum += x;
```

ここで、self は closure または &mut closure 型
である必要がある。
(&closure だと &i32 としてしか取り出せない)

変数のキャプチャーモード

- 変数がクロージャ構造体にキャプチャーされるときの形態は以下の3パターンある

```
struct closure {  
    var1: T1,  
    var2: &mut T2,  
    var3: &T3,  
}
```

クロージャのキャプチャーモード

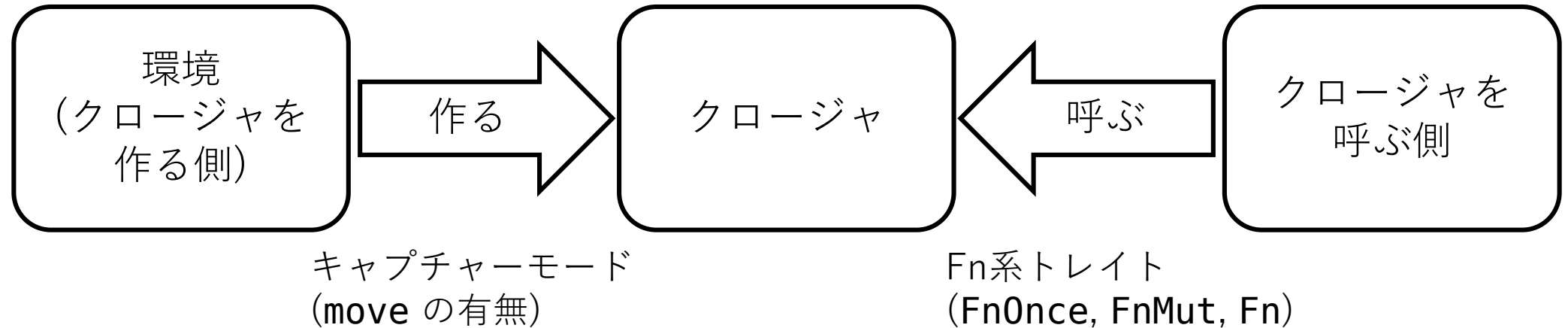
- 本来、キャプチャーモードは変数ごとに決められる。
- Rustでは変数ごとの指定はせず、**既定または move** だけ指定する。

キャプチャーモードの動作

使われ方	既定	move 指定時	実装されるトレイト
ムーブ (借用含む)	T として キャプチャー	T として キャプチャー	FnOnce のみ
&mut 借用 (& 借用を含む)	&mut T として キャプチャー	T として キャプチャー	FnOnce , FnMut
& 借用のみ	&T として キャプチャー	T として キャプチャー	FnOnce , FnMut , Fn

ちょっと複雑なので整理

- 3者の関係、2つのインターフェースに分離できる



ちょっと複雑なので整理

環境
(クロージャを作る側)

- 必ずしも所有権を明け渡さなくてよい。
- クロージャ内での操作が環境に反映される。
- 変数の所有権を取られる。
- クロージャ内での操作が環境に反映されない。

作る

既定のキャプチャー
モード

move キャプチャー

クロージャ

- 環境より長く生きられない。
- 環境に干渉できる。
- **環境より長く生きられる。**
- 環境に干渉しない。

この事情で判断することが多い

ちょっと複雑なので整理

クロージャ

- 外の変数からムーブできる。
(書き込むこともできる)
- 外の変数からムーブできない。
• 外の変数に書き込める。
- 外の変数に書き込めない。
(ムーブもできない)

呼ぶ

FnOnce

FnMut

Fn

クロージャを呼ぶ側

- 1回しか呼べない。
- 順序づければ複数回呼べる。
- 順序なく複数回呼べる。

こちら側の事情で判断することが多い

長生きしたい例

- flat_map 内で map したい時とか

```
fn main() {  
    let v = (0..5)  
        .flat_map(|x| (0..x).map(|_| x))  
        .collect::<Vec<_>>();  
    println!("{:?}", v);  
}
```

このクロージャは長生きする必要がある

長生きしたい例

- flat_map 内で map したい時とか

```
fn main() {  
    let v = (0..5)  
        .flat_map(|x| (0..x).map(move |_| x))  
        .collect::<Vec<_>>();  
    println!("{:?}", v);  
}
```

move をつけて長生きしてもらう

よくあるコンボ

- クロージャが短命すぎると言われる
→ **move** をつける
- すると、或る変数がムーブキャプチャーできないと言われる
→ 仕方ないので **clone** する

明示的なキャプチャー

- 構造体の特定のフィールドだけキャプチャーしたり、変数ごとにキャプチャーモードを変えたいときは、自分で切り分けてから **move** モードでキャプチャーするとよい。

```
let custom_closure = {  
    let field1 = &self.field1;  
    let bar = &bar;  
    move || {  
        // Use field1, bar, baz  
    }  
};
```

※構造体フィールドの自動分割キャプチャーはRFC2229として承認されているが、まだ実装されていない (2018/09)

クロージャと制御フロー

- `return/break/continue` などで外側の制御フローに干渉することはできない。
(`return` と書くとクロージャから `return` する。)
- 以下のようなRubyコードは直訳できない。

```
def shortcutprod(a)
  a.inject(1) do |x, y|
    return 0 if x == 0
    x * y
  end
end
```

Rubyでは関数まで脱出できる。

Rustの `for` はクロージャではないので、脱出が必要なら `for` にオプトできる。
また、`Result` の `FromIterator` 実装などをうまく使うと、イテレーターコンビネーターだけで上手く脱出を模倣できることがある。

イテレータ

イテレータ概要

- イテレータは遅延リストのようなインターフェースを持つ。
- その要素型は、直感に反して参照型となることがある。
- **collect** は多様な宛先型を選べる。これを把握していると使い勝手がかなり良くなる。

イテレータとは

- **Iterator**: 遅延リスト的なインターフェース
- 残りがあれば **Some(x)** を、打ち止めなら **None** を返す
- **ExactSizeIterator**: 残りの長さがわかる
- **DoubleEndedIterator**: 左端だけではなく右端もある。右端から食べることもできる。

IntoIterator

- それ自体はイテレートするのに必要な状態を持たないが、イテレーターに変換することができるデータ。
- 例: **Vec<T>** は「現在位置」の情報を持たないが、現在位置の情報を付加することで **IntoIter<T>** というイテレーターになる。

要素型の怪

```
for i in 0..10 {  
    // i: i32  
}  
for x in vec![0, 1, 2].iter() {  
    // i: &i32  
}
```

要素型の怪

- `Range<i32>` は `&i32` を返せない
 - 仮に返せるとすると、`collect` することで「0が入った参照」「1が入った参照」…… が同時に存在することになる。
 - `Range` にそのための領域は存在しない！
- `Vec<i32>` が `i32` を返すことはできる
 - ……が、それは `i32` が `Clone` であるため。
 - 普通は要素を「見る」だけだから、要素への参照をとるのが自然

要素型の怪

コレクションの式	要素型	備考
<code>start..end</code>	<code>T</code>	<code>T</code> は整数
<code>vec.iter() / &vec</code>	<code>&T</code>	一番よく使う
<code>vec.iter_mut() / &mut vec</code>	<code>&mut T</code>	各要素を編集したいとき
<code>vec.into_iter() / vec</code>	<code>T</code>	Vec の所有権がなくなるので注意
<code>vec.drain(..)</code>	<code>T</code>	指定した範囲の要素を削って取り出す
<code>vec.iter().cloned()</code>	<code>T</code>	複製してるだけ

要素型の怪

- **Iterator** の要素は互いに独立して存在できる
(= 常に **collect** することができる)
- 「単一の **String** バッファに、各行を順番に読み込んでいく」
みたいなのは抽象化できない
こういう抽象化は別途 **streaming_iterator** クレートで提供されている。

collect の魔法

- イテレータをコレクションに収集するには `Iterator::collect` を使う。
- `collect` の収集先は配列だけではなく、色々選べる。

collect の魔法: 基本形

- Vec に収集

```
let data = vec![1, 2, 3];  
let result = data.into_iter().collect::<Vec<_>>();
```

collect の魔法: ハッシュ

- ハッシュマップに収集

```
let data = vec![(1, 4), (2, 8), (5, 7)];  
let result = data.into_iter()  
    .collect::<HashMap<_, _>>();
```

collect の魔法: Result

- 別のコレクションの Result に収集

```
let data = vec![Ok(1), Err(2), Ok(3)];  
let result = data.into_iter()  
    .collect::<Result<Vec<_>, _>>();
```


collect の魔法: 文字列

- 文字を文字列に収集

```
let data = vec!['a', 'b', 'c'];  
let result = data.into_iter()  
    .collect::<String>();
```

練習問題

参考: Rustの練習問題集

- [Rustlings](#): 「既存のコードスニペットを修正して、コンパイルやテストに通す」という形で練習できる。
 - 今回のが難しすぎたら、こっちにするかも
- [Exercism.io](#): 練習問題を解くとメンターがレビューしてくれるサイトらしい。Rustのメンターも[募集中](#)とか。

今日の練習問題

- <https://github.com/qnighy/rust-handson2-exercises>
- 4問

実行方法

- 以下を実行するとテストに失敗します。

```
$ git clone https://github.com/qnighy/rust-handson2-exercises.git  
$ cd rust-handson2-exercises  
$ cargo test
```

練習問題1

- 各行の文字を逆順にしてください。

```
pub fn reverse_each_line(text: &str) -> String {  
    // ...  
}
```

使うときはアンダースコアを外す

練習問題1

- イテレータなので `map` してみる

```
pub fn reverse_each_line(text: &str) -> String {  
    text.split_terminator("¥n").map(|line| line)  
}
```

練習問題1

- 文字のイテレータを取って逆順にする

```
pub fn reverse_each_line(text: &str) -> String {  
    text.split_terminator("¥n")  
        .map(|line| line.chars().rev())  
}
```


練習問題1

- 文字のイテレーターのイテレーターになっているので、フラットにする

```
pub fn reverse_each_line(text: &str) -> String {  
    text.split_terminator("¥n")  
        .flat_map(|line| line.chars().rev())  
}
```

練習問題1

- 各行で改行文字も足すようにする

```
pub fn reverse_each_line(text: &str) -> String {  
    text.split_terminator("¥n")  
        .flat_map(|line| line.chars().rev().chain(Some('¥n')))  
}
```

`std::iter::once` を使うのが面倒なときに `Some` で代用しがち

練習問題1

- 文字のイテレータは文字列にそのまま `collect` できる

```
pub fn reverse_each_line(text: &str) -> String {  
    text.split_terminator("¥n")  
        .flat_map(|line| line.chars().rev().chain(Some('¥n')))  
        .collect()  
}
```

戻り値型にそのまま `collect` するので、型は省略可能

別解

- 行を **for** 文で処理する

```
pub fn reverse_each_line(text: &str) -> String {  
    let mut result = String::new();  
    for line in text.split_terminator("\n") {  
        result.extend(line.chars().rev());  
        result.push_str("\n");  
    }  
    result  
}
```

関数シグネチャの設計

関数シグネチャの設計: 概要だけ

- 参照で受け取り、所有権で返すのが基本
`fn(&str) -> String`
- 単純コピーが可能ならコピー渡しする
`fn(i32) -> String`
- 受け取った参照の一部を返すときは参照返しになる
`fn(&str) -> &str`
- 書き換えられるものは `&mut` 参照で受ける
`fn(&mut Context) -> ()`
- 所有権が絶対必要なときや、ファイナライズ系はムーブ渡し
`fn(Digest) -> [u8; 16]`

型とトレイト

型とトレイト: 概要だけ

- Rustでは型には十分な抽象化能力を与えられていない。
- 型が値の表現をあらわし、トレイトが**型の**性質を表す。
- 「型そのものの性質」であるようなトレイトとしては、 **Eq** や **Default** などがある。
- 間接的に「値の性質」とみなせるトレイトもある。 **Fn** や **Iterator**, **Future**, **Debug**, **Error** など。
- そのようなトレイトは「トレイト安全性を満たす」といい、 **dyn Trait** という型とみなすことができる。

型とトレイト: 概要だけ

- トレイトを型とみなす **dyn Trait** は実行時コストが嵩むため好まれない。
 - また、**Send / Sync** やライフタイムに関する情報が失われるというデメリットもある。
- 逆に、それ以外のケースではトレイトは全てコンパイル時に解決されるため、ほとんどオーバーヘッドがない。(**impl Trait**を含む)
- トレイトは抽象化の数しか存在しないが、型はたくさん作ってよい風潮がある。 **Fn** や **Iterator** はそのような「専用型」のパターンを踏襲しており、HaskellよりC++に近い。

データ構造の設計

データ構造の設計: 概要だけ

- Rustでデータ構造を設計するときの最も特異な性質は「循環参照の扱い」にある。
- 実装レベルでいうと、GCを持たず、RAIIを許可しているため、循環参照との相性が悪い。
 - ユーザーレベルのgcライブラリはあるが、おそらく使い勝手は良くない
- 循環参照は、支配関係が明確でない設計のサイン……かもしれない

データ構造の設計: 概要だけ

- 他言語のパターンにあわせてArc/Weakで設計したくなるが、おそらくこれはイディオマティックではない。
- 支配関係が明確なら、それに沿った参照だけを残して、反対向きのリンクは文脈として持ち回すという方法がある。
- そうではない場合、より大きな親玉のもとで平等に管理するのが正しいかもしれない。
 - しばしば、 `typed_arena` クレートと組み合わせられる。