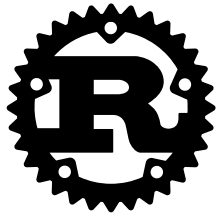
Rust 2019

compscicenter.ru

aleksey.kladov@gmail.com



Лекция 6
Управление Ошибками

Инвалидация Итераторов

```cpp
#include <vector>
#include <iostream>

int main() {
  std::vector<int> xs = {1, 2, 3, 4, 5};
  for (auto x: xs) {
    std::cout << x << std::endl;
    xs.push_back(x);
  }
}
```

Каков результат работы этой программы?

```cpp
#include <vector>
#include <iostream>

int main() {
  std::vector<int> xs = {1, 2, 3, 4, 5};
  for (auto x: xs) {
    std::cout << x << std::endl;
    xs.push_back(x);
  }
}
```

Undefined behavior: `push_back` инвалидирует итераторы

Итератор — пара указателей на конец и начало буфера, `push_back` может переместить буфер

```cpp
#include <vector>
#include <iostream>

int main() {
  std::vector<int> xs = {1, 2, 3, 4, 5};
  for (auto x: xs) {
    std::cout << x << std::endl;
    xs.push_back(x);
  }
}
```

В других языках — не UB, но логическая ошибка или исключение:

- `ConcurrentModificationException`
- dictionary changed sized during iteration

В Rust — ошибка времени компиляции:

```
fn main() {
    let mut xs = vec![1, 2, 3, 4, 5];
    for &x in xs.iter() { (1)
        println!("{}\n", x);
        xs.push(x); (2)
    }
}
```

(1) & ссылка

(2) &**mut** ссылка

```
error[E0502]: cannot borrow `xs` as mutable because it is also
borrowed as immutable
```

Проверка Выхода за Границу

`Index` и `IndexMut` для `[T]` проверяет валидность индекса.

Так как итераторы гарантированно не инвалидируются, то их можно реализовать без проверок

Итератор по `[T]` — пара указателей, так же, как и в C++

В отличие от `[T]`, случайное изменение контейнера во время итерации не ведёт к UB

Стоимость Проверок

Проверка индекса — тривиально предсказуемая ветка, непосредственная цена не значительна

Но проверки могут сломать оптимизации, например, автовекторизацию

Эксперимент

Попробуем просуммировать `N` чисел с векторизацией и без.

Index Sum

```
#[inline(never)] (1)
fn i_sum(xs: &[u32], indexes: &[usize]) -> u32 (2)
{
    let mut sum = 0u32;
    for &idx in indexes {
        let x = xs[idx];
        sum = sum.wrapping_add(x); (3)
    }
    sum
}
```

1. запрещаем `inline`, чтобы не соптимизировать в константу
2. `indexes` — массив `[0, 1, … xs.len()]`, но компилятор об этом не знает (из-за `#[inline(never)]`)
3. используем `wrapping_add` чтобы избежать проверок на переполнение

Index Sum

```
#[inline(never)]
unsafe fn i_sum_unchecked(xs: &[u32], indexes: &[usize]) -> u32 (2)
{
    let mut sum = 0u32;
    for &idx in indexes {
        let x = unsafe { *xs.get_unchecked(idx) }; (1)
        sum = sum.wrapping_add(x);
    }
    sum
}
```

(1) используем `get_unchecked` — **`unsafe`** функцию, не делающую проверку индексов

(2) обязаны пометить всю функцию **`unsafe`**, так как вызывающий код должен гарантировать, что все `indexes` валидны

Генерация Входных Данных

```
fn random(n: usize) -> Vec<u32> {
    let mut r = 92;
    std::iter::repeat_with(move || {
        r ^= r << 13;
        r ^= r >> 17;
        r ^= r << 5;
        r
    }).take(n).collect()
}

fn main() {
    let n: usize = 100_000_000;
    let indexes: Vec<usize> = (0..n).collect();
    let xs: Vec<u32> = random(n);
    ...
}
```

Запуск

```
#[inline(never)]
fn run_benchmark<F: Fn() -> T, T>(name: &str, f: F) -> Vec<T> {
    println!("{}:", name);
    let n = 300;
    let mut res = Vec::with_capacity(n);
    let mut times = Vec::with_capacity(n);
    for _ in 0..n {
        let start = std::time::Instant::now();
        res.push(f());
        times.push(start.elapsed());
    }
    println!("{:?}", times.into_iter().min().unwrap());
    println!("\n");
    res
}
```

Сравнение

```
let r1 = run_benchmark("i_sum", || {
    i_sum(&xs, &indexes)
});

let r2 = run_benchmark("i_sum_unchecked", || {
    i_sum_unchecked(&xs, &indexes)
});

assert_eq!(r1, r2);
```

Результат

`i_sum`	63.63 ms
`i_sum_unchecked`	64.86 ms

Для "скучного" кода ценой `if`, который всегда `false`, можно пренебречь

Sum

```
#[inline(never)]
fn sum(xs: &[u32], lo: usize, hi: usize) -> u32 (1)
{
    let mut sum = 0u32;
    for idx in lo..hi {
        let x = xs[idx];
        sum = sum.wrapping_add(x);
    }
    sum
}
```

(1) `lo = 0`, `hi = xs.len()`, но компилятор про это не знает из-за `#[inline(never)]`

Sum

```
#[inline(never)]
unsafe fn sum_unchecked(xs: &[u32], lo: usize, hi: usize) -> u32 (1)
{
    let mut sum = 0u32;
    for idx in lo..hi {
        let x = unsafe { *xs.get_unchecked(idx) };
        sum = sum.wrapping_add(x);
    }
    sum
}
```

(1) обязаны пометить всю функцию **`unsafe`**, так как вызывающий код должен гарантировать, что `hi <= xs.len()`

Результаты

`sum`	34.51 ms
`sum_unchecked`	17.65 ms

Существенный выигрыш: проверка индексов ломает автовекторизацию

Sum

```
#[inline(never)]
fn sum(xs: &[u32], lo: usize, hi: usize) -> u32 {
    let mut sum = 0u32;
    for &x in xs[lo..hi].iter() { (1)
        sum = sum.wrapping_add(x);
    }
    sum
}
```

(1) делаем проверку один раз, а не на каждой итерации цикла

Как правило, не стоит использовать **`get_unchecked`** для оптимизации: лучше убедить компилятор, что индексы валидны, заменив **`Index`** на **`Iterator`**

Benchmarking Tips

- детерминированный генератор случайных чисел можно просто запомнить: `<< 13; >> 17; << 5;` (xor-shift)
- полезно считать настоящий результат и сравнивать его:
  - компилятор не выкинет код без эффектов
  - защита от очень быстрой, но некорректной реализации
- для коротких CPU-bound программ шум положительный
  ⇒ самая лучшая оценка это минимум
- нужно думать про `inline`

Управление Ошибками

Ошибки Программиста

Следующие ситуации являются ошибками программиста и приводят к панике:

- выход за границу массива
- переполнение типа
- деление на 0
- `assert!(false)`
- `panic!("ooups")`
- ...

Паника: функция вида `fn panic() -> !`

Стратегии паники

Можно настроить, что именно происходит при панике.

- ничего (на embeded устройствах):

```
#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    loop {}
}
```

- `panic=abort` — немедленное завершение процесса (нужна ОС)
- `panic=unwind` — поведение по умолчанию, "чистое" завершение процесса с вызовом деструкторов (разматывание стэка)

Stack Unwinding

```rust
struct Guard(String);

impl Drop for Guard {
    fn drop(&mut self) {
        println!("{}", self.0);
    }
}
```

`Guard` печатает сообщение в `Drop`

Stack Unwinding

```
fn foo() {
    let g = Guard("foo".to_string());
    bar();
}

fn bar() {
    let g = Guard("bar".to_string());
    panic!("boom")
}

fn main() { foo() }
```

```
$ ./main
thread 'main' panicked at 'boom', main.rs:15:5
note: Run with `RUST_BACKTRACE=1` environment variable to display a
backtrace.
bar
foo
```

Resource Acquisition Is Initialization

`Drop` вызывается "всегда" — его удобно использовать для восстановления инвариантов.

- типичный пример — закрыть файловый дескриптор
- нельзя "забыть" вызвать `.close`

Один code path и для штатного завершения, и для ошибок!

RAII vs try-finally

- место использования: definition vs usage
- автоматический `Drop` для агрегатов
- нет привязки к `{}`:

```
fn main() {
    let sock = TcpListener::bind("127.0.0.1:8080").unwrap();
    loop {
        match sock.accept() {
            Ok(stream) => {
                thread::spawn(move || handle_client(stream));
            }
            Err(_) => println!("Error"),
        }
    }
}
```

> *It is funny how people think that the important thing about exceptions is handling them. That is not the important thing about exceptions. In a well-written application there's a ratio of ten to one, in my opinion, of try finally to try catch.*
>
> — *Anders Hejlsberg*

> *Almost all catastrophic failures (92%) are the result of incorrect handling of non-fatal errors explicitly signaled in software.*
>
> — *https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan*

Если **очень** хочется, панику можно перехватить:

- при присоединении потока:

```
let handle = std::thread::spawn(|| panic!("boom"));
match handle.join() {
    Ok(()) => println!("ok"),
    Err(_) => println!("panicked"),
}
```

- где угодно:

```
let result = std::panic::catch_unwind(|| panic!("boom"));
match result {
    Ok(()) => println!("ok"),
    Err(_) => println!("panicked"),
}
```

`thread::panicking` проверяет, есть ли паника

`panic::resume_unwind` переподнимает панику

Границы

Главное в **обработке** ошибок — граница, на которой они обрабатываются:

- процесс (для консольной программы — отличная стратегия)
- поток (= одна задача в пуле потоков)
- запрос (серверы, GUI приложения)

unwinding — приём, который не обязательно использовать именно для ошибок. Тем не менее "don’t use exceptions for flow control" это rule of thumb с большой применимостью

Паника во Время Паники

Паника в `Drop` в процессе разматывание стэка — `abort`

```
impl SafeBufWrite {
    fn flush(&mut self) -> Result { ... }
    fn close(mut self) -> Result {
        self.flush()?;
        mem::forget(self) // обезвредили drop
    }
}

impl Drop for SafeBufWrite {
    fn drop(&mut self) {
        let _ = self.flush(); // игнорируем ошибки
        if !std::thread::panicking() {
            panic!("should be flushed explicitly")
        }
    }
}
```

Result и Option

Паника — критическая ошибка, которую нельзя обработать.

Открытие файла с ошибкой — не "ошибка", ожидаемый результат.

Для моделирования используются типы-суммы, `Result` и `Option`.

```
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

std::fs

```rust
/// Read the entire contents of a file into a string.
fn read_to_string<P>(path: P) -> Result<String, io::Error>
where
    P: AsRef<Path>,
```

std::io

```rust
pub struct Error { ... }

pub enum ErrorKind {
    NotFound,
    PermissionDenied,
    ...
    Other,
}

impl Display for Error { ... }

impl Error {
    fn kind(&self) -> ErrorKind { ... }
}
```

Работа с Result

У `Option` и `Result` большое API:

```
impl Result<T, E> { // Для Option<T> то же самое
    fn unwrap(&self) -> T
    fn expect(&self, msg: &str) -> T
    fn unwrap_or(&self, def: T) -> T
    fn unwrap_or_else<F: FnOnce() -> T>(&self, f: F) -> T
    fn unwrap_or_default(self) -> T

    fn map<U, F: F: FnOnce(T) -> U>(self, op: F) -> Result<U, E>

    fn and_then<U, F>(self, op: F) -> Result<U, E>
    where
        F: FnOnce(T) -> Result<U, E>,
    ...
}

impl<T, E> IntoIterator for Result<T, E> { ... }
```

Работа с Result

```
fn impl Result<T, E> {
    fn ok(self) -> Option<T>;
}

impl Option<T> {
    fn ok_or<E>(self, err: E) -> Result<T, E>
    fn ok_or_else<E, F>(self, err: F) -> Result<T, E>
    where
        F: FnOnce() -> E,

    fn as_ref(&self) -> Option<&T>
    fn as_mut(&mut self) -> Option<&mut T>
}
```

`&Option<T>` — бесполезный тип, используйте **`Option<&T>`**

Примеры

```
trait FromStr {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}

impl str {
    fn parse<F: FromStr>(&self) -> Result<F, <F as FromStr>::Err>
}
```

Метод `parse` позволяет прочитать из строки число, `bool`, путь файловой системы, IP адресс и любой пользовательский тип (return type polymorphism!)

Примеры

`std::io`

```
type Result<T> = std::Result<T, Error>;

struct Error { ... }

pub trait Read {
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

    fn read_to_end(&mut self, buf: &mut Vec<u8>)
    -> Result<usize> { ... }
    fn read_to_string(&mut self, buf: &mut String)
    -> Result<usize> { ... }
    fn read_exact(&mut self, buf: &mut [u8])
    -> Result<()> { ... }
    ...
}
```

`Read` — трейт для чтения потока байт

Quiz

В чём разница?

```
trait Read {
    fn read_to_string(&mut self, buf: &mut String)
    -> Result<usize>
}
```

```
trait Read {
    fn read_to_string(&mut self)
    -> Result<String>
}
```

"Обработка" ошибок

Типичная обработка ошибок:

```
fn read_int<P: AsRef<Path>>(file_path: P) -> io::Result<i32> {
    let mut file = match File::open(file_path) {
        Ok(file) => file,
        Err(err) => return Err(err),
    };
    let mut contents = String::new();
    if let Err(err) = file.read_to_string(&mut contents) {
        return Err(err);
    }
    let n: i32 = match contents.trim().parse() {
        Ok(n) => n,
        Err(err) => {
            return Err(io::Error::new(io::ErrorKind::Other, err));
        }
    };
    Ok(n)
}
```

"Обработка" ошибок

Почти всегда, ошибка пробрасывается наверх, хочется это автоматизировать.

?

`expr?` рассахаривается в **`match`**:

```
match expr {
    Ok(ok) => ok,
    Err(err) => return Err(err),
}
```

```
fn read_int<P: AsRef<Path>>(file_path: P) -> io::Result<i32> {
    let mut file = File::open(file_path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let n: i32 = contents.trim().parse().map_err(|err| {
        Err(io::Error::new(io::ErrorKind::Other, err))
    })?;
    Ok(n)
}
```

Quiz

В чём разница?

```
trait Read {
    fn read_to_string(&mut self, buf: &mut String)
    -> Result<usize>
}
```

```
trait Read {
    fn read_to_string(&mut self)
    -> Result<String>
}
```

Quiz

Первый вариант позволяет переиспользовать аллокацию:

```
let mut buf = String::new();
for path in paths {
    buf.clear(); // очищаем содержимое, сохраняем память
    let file = File::open(path)?;
    file.read_to_string(&mut buf)?;
    process(&buf);
}
```

Конверсии

Частый паттерн — оборачивание ошибок низкого уровня в ошибки высокого уровня:

```
enum DbError {
    Io(io::Error),
    QuerySyntaxError(QuerySyntaxErrorError),
    ConstraintViolation(ConstraintViolation),
    ...
}
```

В Rust достигается при помощи ? и трейтов для конверсий

Конверсии

`Deref` — лекция 3, автоматически работает для оператора `.`

`AsRef` — преобразование ссылок

```
trait AsRef<T>
    fn as_ref(&self) -> &T;
}

impl AsRef<Path> for str {
    ...
}
```

`Borrow` — `AsRef` + согласованные `Eq`, `Ord`, `Hash`

```
trait Borrow<Borrowed> {
    fn borrow(&self) -> &Borrowed;
}
```

Конверсии

```
trait From<T> {
    fn from(T) -> Self;
}

trait Into<T> {
    fn into(self) -> T;
}

impl<T, U> Into<U> for T where U: From<T> {
    fn into(self) -> U {
        U::from(self)
    }
}
```

Произвольное преобразование типов

Реализовывать надо `From`, использовать удобно `Into`

?

```
enum MyError {
    Io(std::io::Error),
    Parse(std::num::ParseIntError),
}

...

fn read_int<P: AsRef<Path>>(file_path: P) -> Result<i32, MyError>
{
    let mut file = File::open(file_path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let n: i32 = contents.trim().parse()?;
    Ok(n)
}
```

? использует `From` чтобы преобразовать тип ошибки

?

```
enum MyError {
    Io(std::io::Error),
    Parse(std::num::ParseIntError),
}

impl From<std::io::Error> for MyError {
    fn from(err: std::io::Error) -> MyError {
        MyError::Io(err)
    }
}

impl From<std::num::ParseIntError> for MyError {
    fn from(err: std::num::ParseIntError) -> MyError {
        MyError::Parse(err)
    }
}
```

?

Бонус: `?` работает с `Option<T>`

? vs Exceptions

- `?` требует явной пометки на call site (C, Go, Swift)
- возвращаемый тип поменялся с `T` на `Result<T>` — нужно исправлять все места вызова
- "исключение" — честный тип, работает с HoF
- сложно управлять иерархиями ошибок (много `From`)

Интересный middle ground: один тип ошибки + касты (Swift, Midori)

unwinding или возвращаемое значение — implementation detail

Итоги

- нужно отличать ошибки программиста от ошибок окружения
- ошибки редкий code path, обработка ошибок плохо тестируется
- инварианты (выполняется всегда) лучше явной обработки ошибок
- главный вопрос при перехвате ошибки — "где?" (границы)

http://joeduffyblog.com/2016/02/07/the-error-model/

Где std::error::Error?

В Rust есть стандартный трейт для ошибок: `Error`.

Использовать его не обязательно, и в нём есть несколько проблем.

Подробности — в лекции про dynamic dispatch