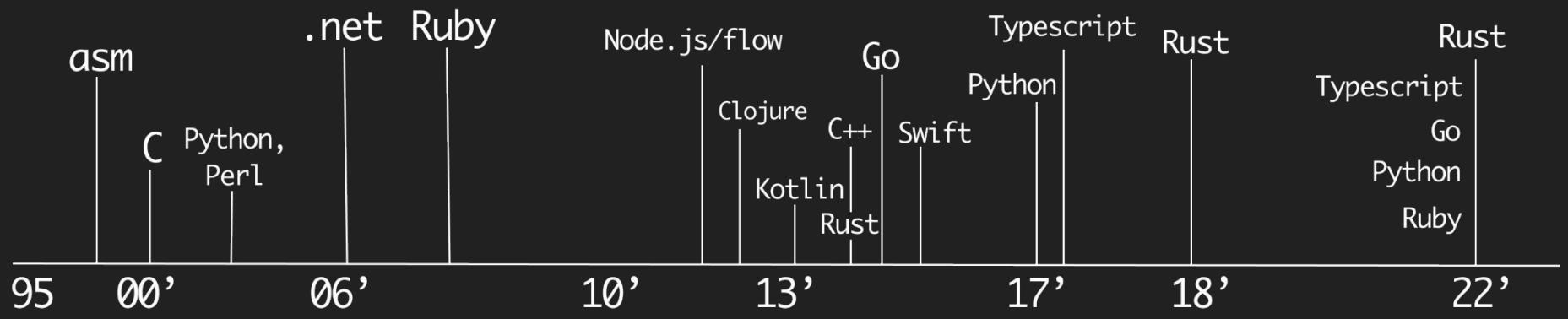
Rust Idioms & Patterns

@jondot 🦀

{twitter,github}.com/jondot 👋

- CEO @ Spectral → Head of Dev-First Security @ Check Point
- CTO HiredScore, Klarna, Como/Conduit
- Team Reversim 🎉
- Programming language freak:



Idioms and patterns

- Are opinionated, pragmatic usage of a language, tackling:
  - Easy vs Hard
  - Readability + acceptable/correct use
  - Creativity 😃 ← ? → 😕 Fanaticism?
- It’s OK to find your own + not mandatory, but be mindful of what’s out there.

What's out there, per language?

Ruby	Javascript	Typescript	Java	Rust	Go
A LOT	A good amount	Almost none	A good amount	Medium sized	Very little
Freedom & happiness	Browser wars	???	Backwards comp. + design overhead	Evolution?	Small lang.

Idioms In Rust: What's the goal?

- Rust is a safe, powerful, cost-aware, and <u>unforgiving</u> systems programming language
- We want to get at:
  - Readable, maintainable code
  - Use Rust to its full potential: expressive, cost-efficient and safe code
  - Have a happy compiler
  - Giving others more forgiving APIs

How?

- Abide to conventions and also through tooling
- Understand design principles that are unique to Rust
  - And that are repeating
- Use and track the *evolving* knowledge from the community, and standard Rust “as intended”

Giving others more forgiving APIs*

```rust
fn foo_bad(text: String){

}
fn foo_good(text: &str){

}
```

```rust
fn foos_bad(text: Vec<u8>){

}
fn foos_good(text: &[u8]){

}
```

Cost, ownership, + accepting more types

*You'll notice, oddly enough, SOLID take more focus than not in good Rust code

Conventions

Don’t worry - use clippy. Really, use it <u>today</u>

```
let _n: u64 = match opt {
    Some(n: Result<u64, String>) => match n {
        Ok(n: u64) => n,
        _ => return,
    },    this `match` can be collapsed into the outer `match`
    None => return,
};
```

```
Some(n) => match n {
     ^ replace this binding
    Ok(n) => n,
    ^^^^^ with this pattern
```

```
1 implementation
struct Foo;
0 implementations
struct Bar(Foo);
impl Foo {
    // Bad. The type
    fn new() -> Bar {
        Bar(Foo{})
    }
}
```

```
methods called `new` usually return `Self`
`#[warn(clippy::new_ret_no_self)]`on by def
for further information visit https://rust-
lang.github.io/rust-
clippy/master/index.html#new_ret_no_self cli
View Problem (⇧⌘J)   No quick fixes available
associated function is never used: `new
```

```
let x: f64 = 3.14;    approximate value of `f{32, 64}::consts::PI`
```

```
if let Some(2) = x {
    do_thing();
}
```

a

```
if x == Some(2) {
    do_thing();
}
```

b

Opt-in to show more

clippy

Category	Description	Default level
`clippy::all`	all lints that are on by default (correctness, suspicious, style, complexity, perf)	warn/deny
`clippy::correctness`	code that is outright wrong or useless	deny
`clippy::suspicious`	code that is most likely wrong or useless	warn
`clippy::style`	code that should be written in a more idiomatic way	warn
`clippy::complexity`	code that does something simple but in a complex way	warn
`clippy::perf`	code that can be written to run faster	warn
`clippy::pedantic`	lints which are rather strict or have occasional false positives	allow → warn
`clippy::nursery`	new lints that are still under development	allow → warn
`clippy::cargo`	lints for the cargo manifest	allow

rustc

		uninhabited-static, unstable-name-collisions, unsupported-calling-conventions, where-clauses-object-safety
nonstandard-style	Violation of standard naming conventions	non-camel-case-types, non-snake-case, non-upper-case-globals
rust-2018-compatibility	Lints used to transition code from the 2015 edition to 2018	absolute-paths-not-starting-with-crate, anonymous-parameters, keyword-idents, tyvar-behind-raw-pointer
rust-2018-idioms	Lints to nudge you toward idiomatic features of Rust 2018	bare-trait-objects, elided-lifetimes-in-paths, ellipsis-inclusive-range-patterns, explicit-outlives-requirements, unused-extern-crates → warn
rust-2021-compatibility	Lints used to transition code from the 2018 edition to 2021	array-into-iter, bare-trait-objects, ellipsis-inclusive-range-patterns, non-fmt-panics, rust-2021-incompatible-closure-captures, rust-2021-incompatible-or-patterns, rust-2021-prefixes-incompatible-syntax, rust-2021-prelude-collisions → warn
	Lints that	dead-code, path-statements, redundant-semicolons, unreachable-code, unreachable-patterns, unused-

```
$ cargo clippy -- \
  -W clippy::nursery \
  -W clippy::pedantic \
  -W rust-2018-idioms \
  -W rust-2021-compatibility
```

```
"rust-analyzer.checkOnSave.command": "clippy",
"rust-analyzer.checkOnSave.extraArgs": [
  "--",
  "-W",
  "clippy::pedantic",
  "-W",
  "clippy::nursery",
  "-W",
  "rust-2018-idioms",
],
```

Per module/file:

```
#![warn(clippy::pedantic)]
```

Use Rust's naming conventions

Method name	Parameters	Notes	Examples
`new`	no self, usually ≥ 1[^1]	Constructor, also cf. `Default`	`Box::new`, `std::net::Ipv4Addr::new`
`with_...`	no self, ≥ 1	Alternative constructors	`Vec::with_capacity`, `regex::Regex::with_size_limit`
`from_...`	1	cf. conversion traits	`String::from_utf8_lossy`
`as_...`	`&self`	Free conversion, gives a view into data	`str::as_bytes`, `uuid::Uuid::as_bytes`
`to_...`	`&self`	Expensive conversion	`str::to_string`, `std::path::Path::to_str`
`into_...`	`self` (*consumes*)	Potentially expensive conversion, cf. conversion traits	`std::fs::File::into_raw_fd`

Traits
Verb > noun > adj.

"Copy", "Clone", "Serialize"

Errors
[Verb][Object][Error]
ParseAddrError

Beware of “Attack of the clones”, prefer by ref

- Clone may or may not be costly, **prefer using references+lifetimes** on arguments
- Overusing clone is tempting because copies of data avoids borrow.
- It’s not a sin, but a place to find optimizations if you need it

```
return Foo {
    path: path.clone(),
    addr: addr.clone(),
    place: place.clone()
}
```

- Exception: smart pointers clones (**`Arc`**, **`Rc`**, etc.)

```
let data: Arc<Mutex<&str>> = Arc::new(data: Mutex::new("foo"));
for _ in 0..10 {
    let data: Arc<Mutex<&str>> = data.clone();
```

Design

<u>Do upfront design rule #1:</u>
Careful with rabbit holes, they end up as a trap

- GC languages make Rabbit holes OK: GC is *accountable* for your decisions, hiding/delaying bad design decisions (which Rust will not allow)

- “I’ll just code as I go & refactor, Rust is static!” until you hit the borrow checker wall, and:
  - Refactor ownership, which requires
  - Refactoring your object model, which means
  - Refactoring logic, which makes you doubt your
  - Modules and API surface (visibility)
- “Fine!, I’ll do TDD, refactoring guided by tests!” until you discover you can’t “see” ownership from down below

Canonical example: Borrow surface-area that’s too wide

Bottom up thinking →

Didn’t see this coming →

Better, also, “makes sense” →

```rust
pub struct Queue {
    pub path: String,
    pub settings: String,
    pub num_jobs: u32,
    pub retry_delay: u32,
    pub num_retry: u32,
}
fn use_stuff(){
    let q = Queue::new();

    // borrow checker twisties
    // returns part of q as a reference, yikes!.
    let rq = f1(&q);
    f2(&q); // can't use q
    println!("{}", rq)
}
```

You, 1 second ago | 1 author (You) | 0 implementations

```rust
pub struct Queue {
    config: QueueSettings,
    job: JobSettings,
}
```

- Going stateless or functional is a safe bet
- Prefer single purpose over “do it alls” big surface area, big responsibility (ISP?)
- Do <u>significant design</u> before coding, but not “big design upfront”, nail responsibilities and ownership on a piece of paper

Do upfront design rule #2

Design concurrent code upfront

- Rust is unforgiving for state that can't be shared safely

  *At any given time, you can have either **one mutable reference** or any **number of immutable references***

- Shared state: don't have shared state. In doubt prefer immutable, read-only
- Don't reach out directly to threads!
- For collections, use `rayon`
- For coordination use `mpsc`
- For shared state: `Arc::new(Mutex::new(T))`
  - We move stuff into threads because it may outlive our code
  - Atomic Reference-Counted (ARC), mutex are not copy + Rc is not thread-safe
  - Mutex lock your shared state access
  - Rust is happy

rayon → **mpsc**

```rust
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter() // <-- just change that!
        .map(|&i| i * i)
        .sum()
}
```

```rust
let (tx, rx) = sync_channel::<i32>(0);
thread::spawn(move|| {
    tx.send(53).unwrap();
});
rx.recv().unwrap();
```

Yea, ok, well, **`Arc<Mutex>`** and threads

```rust
let data: Arc<Mutex<&str>> = Arc::new(data: Mutex::new("foo"));
for _ in 0..10 {
    let data: Arc<Mutex<&str>> = data.clone();
    thread::spawn(move || {
        let mut _data: MutexGuard<&str> = data.lock().unwrap();
    });
}
```

Prefer expressions over binding & flow control

- Shares FP traits (law of substitution): expand or contract are mechanical operations (*almost) given that everything is an expression
- Variable hygiene + scoping **mut**
- When everything is an expression it is easier to fold, refactor as code evolves

```rust
let data: Value = if let Some(path) = cli.data_path {
    let content: Vec<u8> = tokio::fs::read(path).await?;
    serde_json::from_slice(&content)?
} else if let Some(data: Value) = cli.data_value {
    data
} else {
    serde_json::json!({})
};
```

Mechanical folding

```
match foo {
    a => {}
    b => {}
    c => {}
}
```

```
match foo {
    a => {}
    b => {}
    c => {
        match d {
            e => {}
            f => {}
        }
    }
}
```

```
match foo {
    a => {}
    b => {}
    c => {}
    e => {}
    f => {}
}
```

```
let c = match a {
    Some(a) => {
        match b {
            Some(b) => {},
            None => {},
        }
    }
    None => {
        match b {
            Some(b) => {},
            None => {},
        }
    }
};
```

```
let c = match (a, b) {
    (Some(a), Some(b)) => {},
    (Some(a), None) => {},
    (None, Some(b)) => {},
    (None, None) => {},
};
```

Synthetic expressions

```
let res = match (cond_1(), cond_2()) {
    (false, false) => {},
    (false, true) => {},
    (true, false) => {},
    (true, true) => {},
} ;
```

Scopes for temporary mut > shadowing rebind

```
let data = {
    let mut data = get_vec();
    data.sort();
    data
};
```

Traits as OOP iface/Polymorphism/Dyn.dispatch

```rust
struct Runner {
    fs: // <something that looks like a FileSystem and implements #rmdir>
}
```

```rust
let r: Runner = Runner {
    //it's enough that fs will be &dyn FileSystem.
    fs: RealFS::new()
};
```

```rust
impl Runner {
    fn new(which_one: &str)->Self{
        Runner {
            fs: hmm..//<???>
        }
    }
}
```

Traits as OOP iface/Polymorphism/Dyn.dispatch

- Dynamic dispatch vs `impl Trait`: when you can't represent all possible impl's - `dyn` marks the spot.
- But the magic of dyn dispatch is late decision about concrete type, so we're borked.
- Solve with this recipe:

1. Identify a single owner
2. Give it a field `fs: Box<dyn FileSystem>`
3. Pass it around: `other_foo(self.fs.as_ref())`
4. Receiver accepts `&'a dyn FileSystem`

All together now

```rust
pub struct Runner {
    git: Box<dyn GitProvider>,
}
```

```rust
impl Default for Runner {
    fn default() -> Self {
        Runner {
            git: Box::new(GitCmd::default()),
        }
    }
}
```

```rust
let sl = Shortlink::new(&config, self.git.as_ref());
```

```rust
impl<'a> Shortlink<'a> {
    pub fn new(config: &'a Config, git: &'a dyn GitProvider) -> Self {
        Self { config, git }
    }
```

Built ins

Use `default` more

- Rust doesn't have "official" constructors, but 2 idioms: `default`, `new`
- Use `default` for no parameters, `new` otherwise
- Compared to `` `new` ``, `default` has some advantages:
  - `_or_default()` (`Option::unwrap_or_default()`)
  - `{foo: "foo", ..Foo::default()}` (literal initialization)
- Default is transitive
  - `#[derive(Default)]` works only if all fields implement it as well

Give structs some Trait candy

- Make your structs more useful, by implementing <u>all</u> of these:
  - **`Debug, Display`**
  - **`Default`**
  - **`Serialize + Deserialize`**
  - **`Clone`**
  - **`PartialOrd, PartialEq, Hash`**

Work with `Option`'s builtins

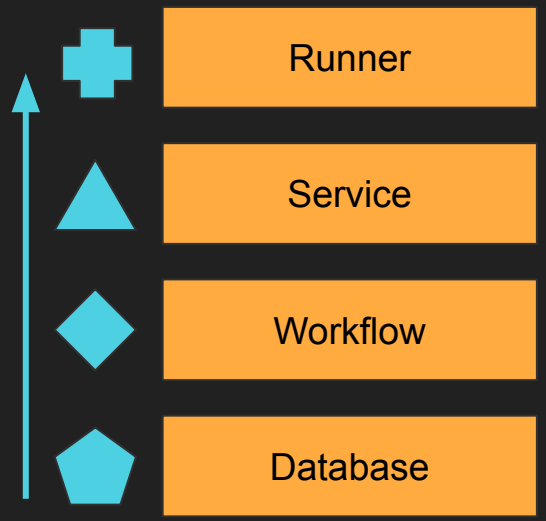
- Signals “has nothing”, and is different from `Result`
- `.and_then` as a ‘singular’ map for piping operations and chain access
- `result.ok()` to move to Option
- `opt.ok_or_else(|| { <error> } )` to move to `Result`
- Is an **iterator (collection of one)**, flatten, filter on “Some” values, `.iter()` too.

```
let a: [&str; 5] = ["1", "two", "NaN", "four", "5"];
let mut iter: Vec<i32> = a [&str; 5]
    .iter() Iter<&str>
    .filter_map(|s: &&str| s.parse().ok()) impl Iterator<Item = i32>
    .collect::<Vec<i32>>();
```

Use **anyhow** for error handling

- Per layer:
  - Stop
  - Convert error manually
  - Propagate
- Annoying: can't use '?' properly

- Error propagation libs struggled because std Error was imperfect. It's fixed now (RFC2504).
- `**anyhow**` takes advantage of that.
  - Keeps track of the error chain
  - Supports downcasting + conversions
  - Suports std Error so: it works + future proof



```rust
fn get_ref_or_default(&self, location: &Location) -> Result<RemoteInfo> {
    let refs = self.ls_remote(location)?;
    if let Some(ref gref) = location.gref {
        let ref_ = refs
            .iter()
            .find(|r| r.ref_.ends_with(gref))
            .ok_or_else(|| anyhow::anyhow!("no such ref found: {}", gref))?;
        return Ok(ref_.clone());
    }

    let head = refs
        .iter()
        .find(|r| r.ref_ == "HEAD")
        .ok_or_else(|| anyhow::anyhow!("no HEAD ref found"))?;
    let default_branch = refs
        .iter()
        .find(|r| r.ref_ != "HEAD" && r.revision == head.revision)
        .ok_or_else(|| anyhow::anyhow!("no default branch found"))?;
    Ok(default_branch.clone())
}
```

Don’t panic!

- Don’t **`panic!`** as an error handler, only when it’s really a stop-the-world.
- Add `` `.context()` `` to errors with **`anyhow`** + use **`bail!`**, **`anyhow!`**, **`ensure!`**
- Results have some extras when viewed as a collection:

```
let strings: Vec<&str> = vec!["tofu", "93", "18"];
let numbers: Vec<_> = strings Vec<&str>
    .into_iter() IntoIter<&str>
    .filter_map(|s: &str| s.parse::<i32>().ok()) in
    .collect();
```

```
let numbers: Result<Vec<_>, _> = strings
.into_iter()
.map(|s| s.parse::<i32>())
.collect();
```

Thanks!

github.com/**rusty-ferris-club**

```rust
{
    let eventually_builtins: Arc<OnceCell<LoadedBuiltins>> = eventually_builtins.clone();
    linker.func_wrap3_async(
        "env",
        "opa_builtin1",
        move |caller: Caller<'_, _>, builtin_id: i32, _ctx: i32, param1: i32| {
            let eventually_builtins: Arc<OnceCell<LoadedBuiltins>> = eventually_builtins.
            Box::new(async move {
                eventually_builtins Arc<OnceCell<LoadedBuiltins>>
                    .get() Option<&LoadedBuiltins>
                    .expect(msg: "builtins where never initialized") &LoadedBuiltins
                    .builtin(caller, &memory, builtin_id, args: [param1]) impl Future<Outp
                    .await
            })
        },
    )?;
}

{
    let eventually_builtins: Arc<OnceCell<LoadedBuiltins>> = eventually_builtins.clone();
    linker.func_wrap4_async(
        "env",
        "opa_builtin2"
```

Conversions

- Implement ‘from’, never ‘into’
- Into<Option<T>>

Into_iter vs Iter

- The iterator returned by into_iter may yield any of T, &T or &mut T, depending on the context.
- The iterator returned by iter will yield &T, by convention.
- The iterator returned by iter_mut will yield &mut T, by convention.