The Three R's: Redis, Rust & Raft

Uri Shachar, Director of Software Engineering, Redis

redis ENGINEERING

Introductions

Uri Shachar

Senior Director of Software Engineering, Redis.

redis ENGINEERING

Agenda

1. Redis - The OSS & The company
2. Rust @ Redis
3. Replacing Redis Cluster with Raft based mechanism

Our Roots are in Open Source

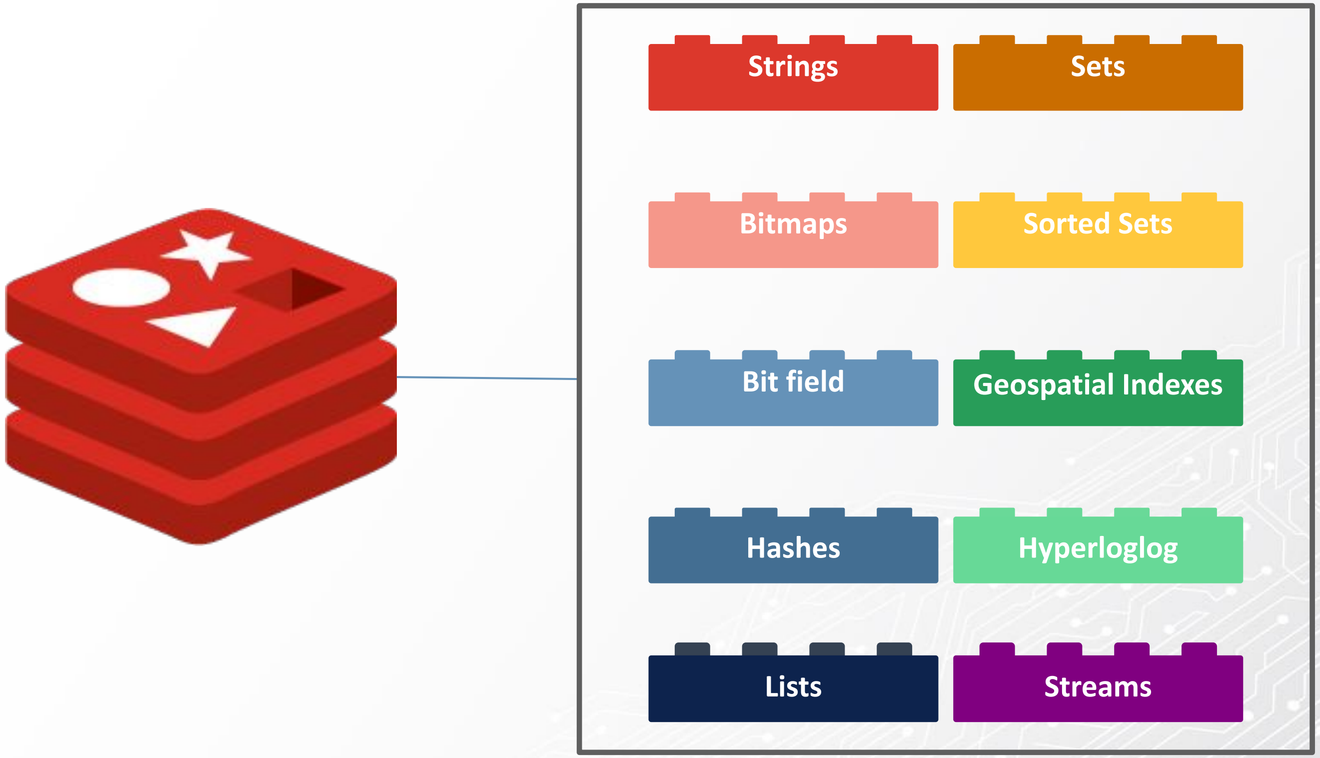
An **In-memory open source database**, supporting a variety high performance operational, analytics and hybrid use cases

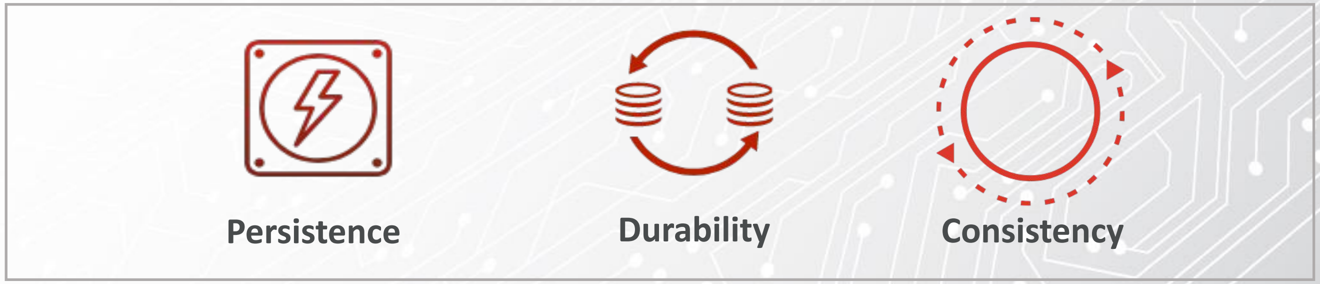
Redis OSS

Commonly used for:

- Cache
- Message bus (Pub/Sub)
- Session store
- Leaderboard
- Distributed Lock
- Job Queue
- …

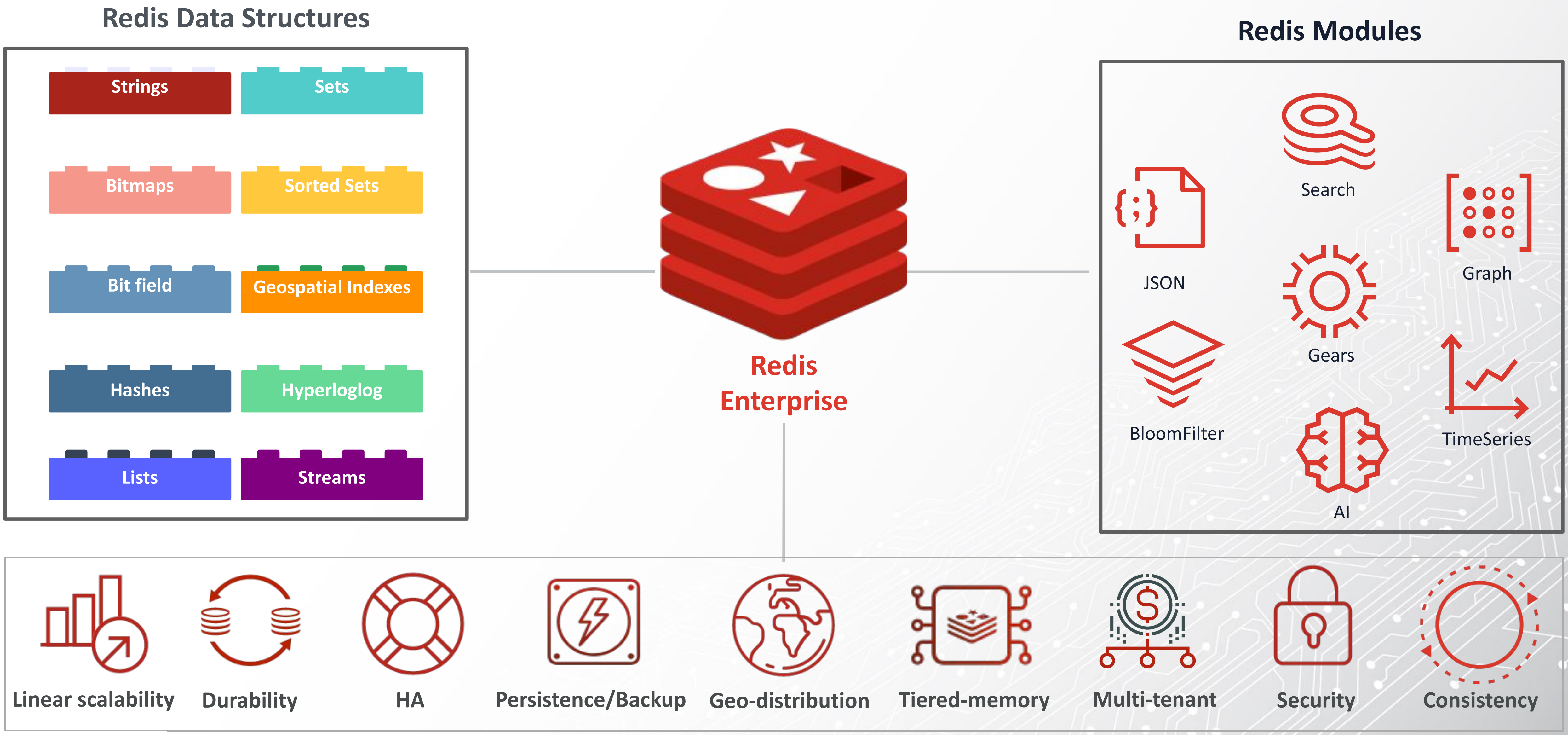
Redis Data Structures





redis ENGINEERING

Redis



redis ENGINEERING

Available Everywhere

On major CSPs (either directly or via partnerships) as a managed product.

And as software.

Google Cloud

Amazon Web Services

Microsoft Azure

VMWare Tanzu

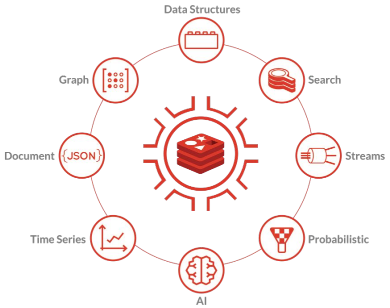
Kubernetes

RedHat OpenShift

Bare Metal / VM

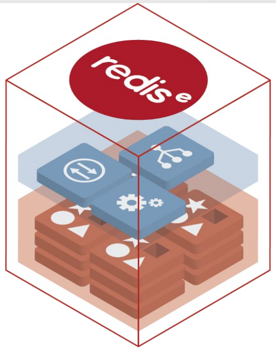
redis ENGINEERING

Rust @ Redis



Modules

- Redis module interface
- ReJSON
- Other



Control plane

- New projects
- Quorum management
- Authentication

Core

- New Redis Clustering
- Client libs (redis-rs)
- More coming…

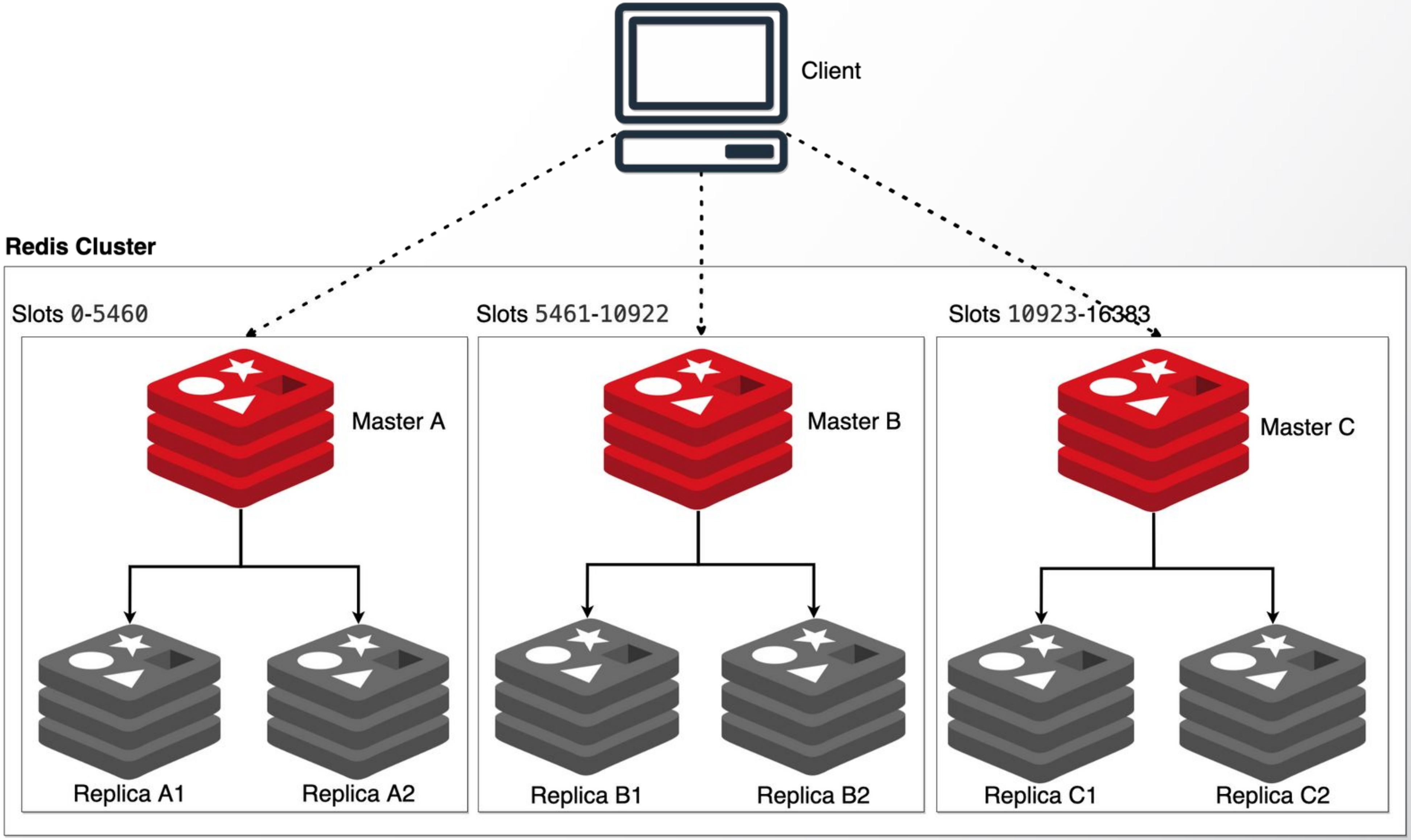
Rust @ Redis

Specifically, my team is working almost entirely in Rust:

- K8s Operators for managing Redis at scale
- Control plane processes for managing and monitoring Redis instances
- Component testing framework & tests
- New Redis Clustering implementation

redis ENGINEERING

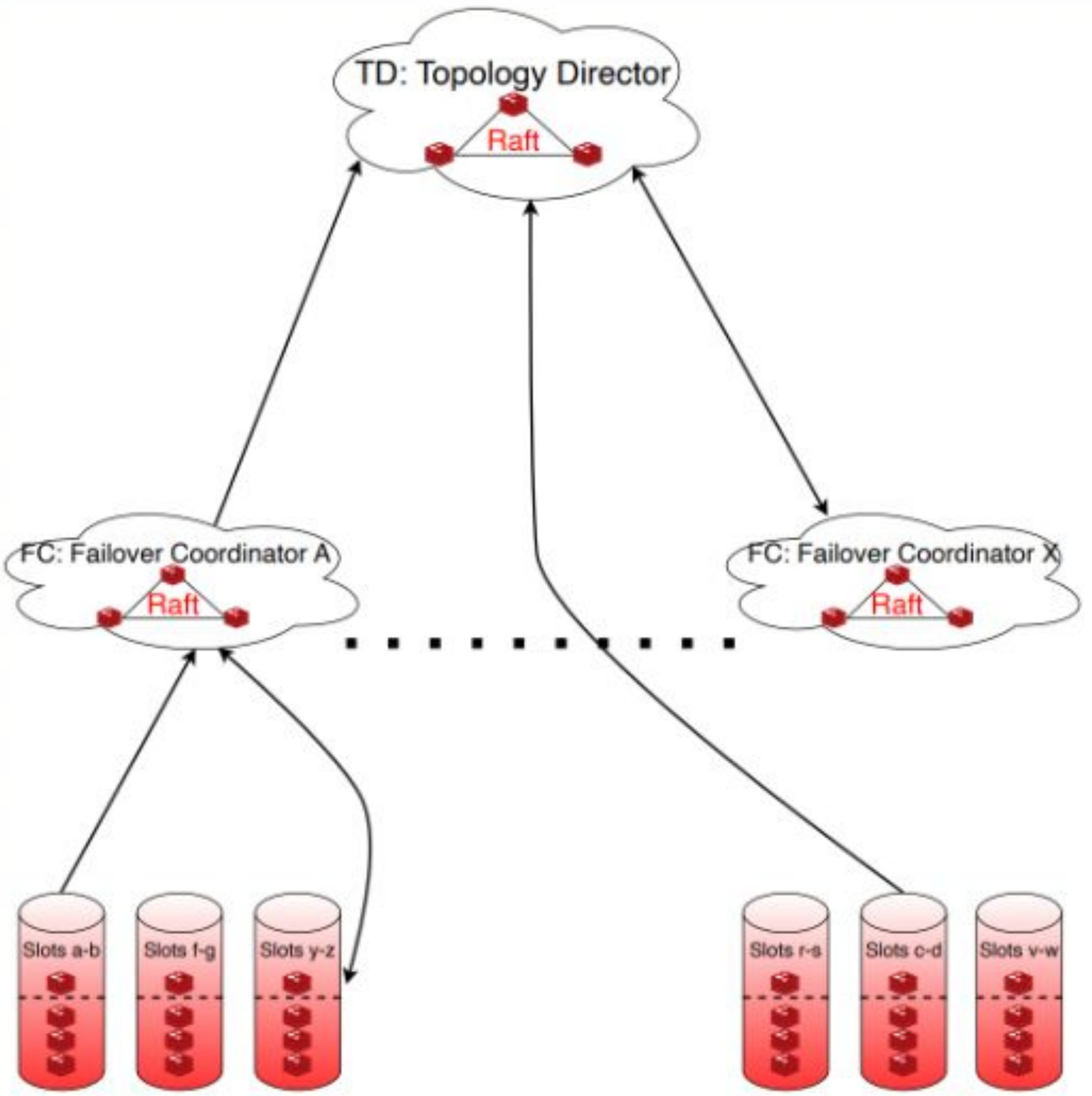
Redis Clustering



redis ENGINEERING

Flotilla - New Redis Cluster

- Two tiers of strongly consistent, consensus-based control plane systems
- Strong Consistency using Raft
- Implemented as a Rust module (of course :))
- Coming soon to Open Source near you



redis ENGINEERING

Thank you

Uri Shachar

redis
ENGINEERING

Macros in Rust

Unleashing the power of Metaprogramming

Yael Tzirulnikov/Software engineer, redis

Introductions

Yael Tzirulnikov

Senior Software Engineer

A Riddle / C.GPT

I’m short and sweet, but can be tricky at times.
I can save you a lot of typing lines.
I help simplify code that’s long and repetitive,
Just call my name, and you’ll be so effusive!
What am I?

A Macro!

Agenda

1. Macros in general
2. Declarative macros
3. Procedural macros
4. Some cool macros

redis
ENGINEERING

Macros in general

- Meta programming - code that writes other code
- Can receive a variable number of parameters
- Macros are expanded before the program finishes compiling
- More complexity- hard to read and maintain

redis ENGINEERING

Types of Macros in Rust

Declarative Macros

- `macro_rules!`

Procedural Macros

- `#[proc_macro]`
- `#[proc_macro_derive]`
- `#[proc_macro_attribute]`

redis ENGINEERING

Declarative macros

- Allow you to write something similar to match expression
- Macros also compare a value to patterns that are associated with particular code:
  - the value is the literal Rust source code passed to the macro
  - the patterns are compared with the structure of that source code
  - the code associated with each pattern, when matched, replaces the code passed to the macro

Example- vec!

```
let v: Vec<u32> = vec![1, 2, 3];
```

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

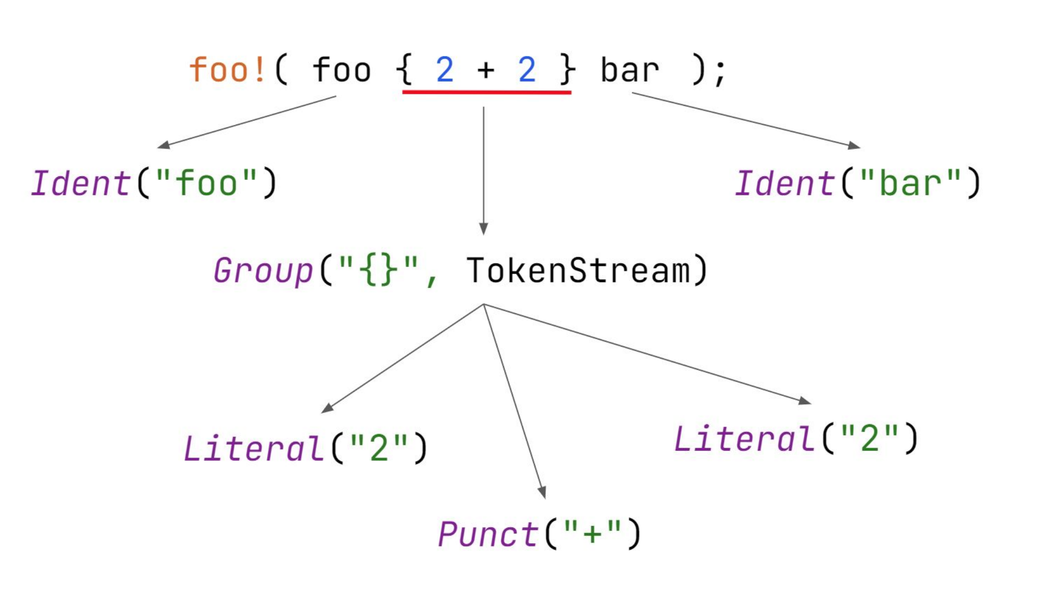
```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

Procedural macros

- Act more like functions
- Accept some code as an input, operate on that code, and produce some code as an output

```
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
}
```



Syn, Quote crates

Syn:
Parse a stream of Rust tokens into a syntax tree of Rust source code

Quote:
Turn Rust syntax tree data structures into tokens of source code

Procedural macros types

- custom derive
- attribute-like
- function-like

Custom derive macros

```
#[derive(serde::Serialize, serde::Deserialize)]
pub struct Module {}
```

- Used on structs and enums
- Specify code **added** with the derive attribute

redis ENGINEERING

Custom derive macro- example

```rust
impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}
```

↓

```rust
#[derive(HelloMacro)]
struct Pancakes;
```

redis ENGINEERING

hello_macro_derive crate

```toml
[package]
name = "hello_macro_derive"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

redis ENGINEERING

hello_macro_derive implementation

```
extern crate proc_macro;

use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast : DeriveInput = syn::parse( tokens: input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}
```

Syn result

```
DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

redis ENGINEERING

hello_macro_derive implementation cont.

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name : &Ident = &ast.ident;
    let gen : TokenStream = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

The result

```rust
#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Attribute like macros

```
#[route(GET, "/")]
fn index() {
```

- define custom attributes usable on any item
- The returned TokenStream **replaces** the item with an arbitrary number of items.

The signature of the macro definition function would look like this:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

redis ENGINEERING

Function like macros

```
todo_or_die::after_date!(2023, 3, 1);
```

```
let testing = indoc! {"
        def hello():
            print('Hello, world!')

        hello()
    "};
```

- Look like function calls but operate on the specified tokens
- Similar to macro_rules! macros, but much more flexible
- The output TokenStream **replaces** the entire macro invocation.

```
#[proc_macro]
pub fn after_date(input: TokenStream) -> TokenStream {}
```

redis ENGINEERING

Some cool macros

Log-derive- Result logging

https://crates.io/crates/log-derive

```
#[logfn(Err = "Error", fmt = "Failed Sending Packet: {:?}")]
#[logfn_inputs(Info)]
 fn send_hi(addr: SocketAddr) -> Result<(), io::Error> {
     let mut stream = TcpStream::connect(addr)?;
     stream.write(b"Hi!")?;
     Ok( () )
 }
```

A macro to log errors and inputs from a function.

redis ENGINEERING

Some cool macros

Recap- regex parsing

https://crates.io/crates/recap

```
#[recap(regex = r#"(?x)
(?P<foo>\d+)
\s+
(?P<bar>true|false)
\s+
(?P<baz>\S+)
"#)]
struct LogEntry {
    foo: usize,
    bar: bool,
    baz: String,
}
```

An easy way to build data from regex strings!

```
let entry: LogEntry = "1 true hello".parse()?;
```

redis ENGINEERING

Some cool macros

Shrinkwraprs — generate distinct types

https://crates.io/crates/shrinkwraprs

```
#[derive(Shrinkwrap)]
struct Email(String);
```

```
let email = Email("chiya+snacks@natsumeya.jp".into());
let is_discriminated_email =  email.contains("+");  // Woohoo, we can
use the email like a string!
```

Shrinkwraprs redefines a datatype as a new distinct type. You can add the Shrinkwrap attribute to inherit all the behaviour of the embedded datatype.

redis ENGINEERING

Some cool macros

Metered

https://crates.io/crates/metered

This macro will automatically generate the following stats on a method:

- `HitCount`: number of times a piece of code was called
- `ErrorCount`: number of errors returned – (works on any expression returning a Result)
- `InFlight`: number of requests active
- `ResponseTime`: statistics on the duration of an expression
- `Throughput`: how many times an expression is called per second.

redis ENGINEERING

Some cool macros

Metered

```
#[metered(registry = BizMetrics)]
impl Biz {
    #[measure([HitCount, Throughput])]
    pub fn biz(&self) {
        let delay =
std::time::Duration::from_millis(rand::random::<u64>() %
200);
        std::thread::sleep(delay);
    }
}
```

Retrieve the metrics as serialised yaml:

```
let biz = Arc::new(Biz::default());
let serialized =
serde_yaml::to_string(&*biz).unwrap()
;
```


```
metrics:
 biz:
   hit_count: 1000
   throughput:
     - samples: 20
       min: 35
       max: 58
       mean: 49.75
       stdev: 5.146600819958742
       90%ile: 55
       95%ile: 55
       99%ile: 58
       99.9%ile: 58
       99.99%ile: 58
     - ~
```


redis ENGINEERING

Future

There are some strange edge cases with `macro_rules!`.

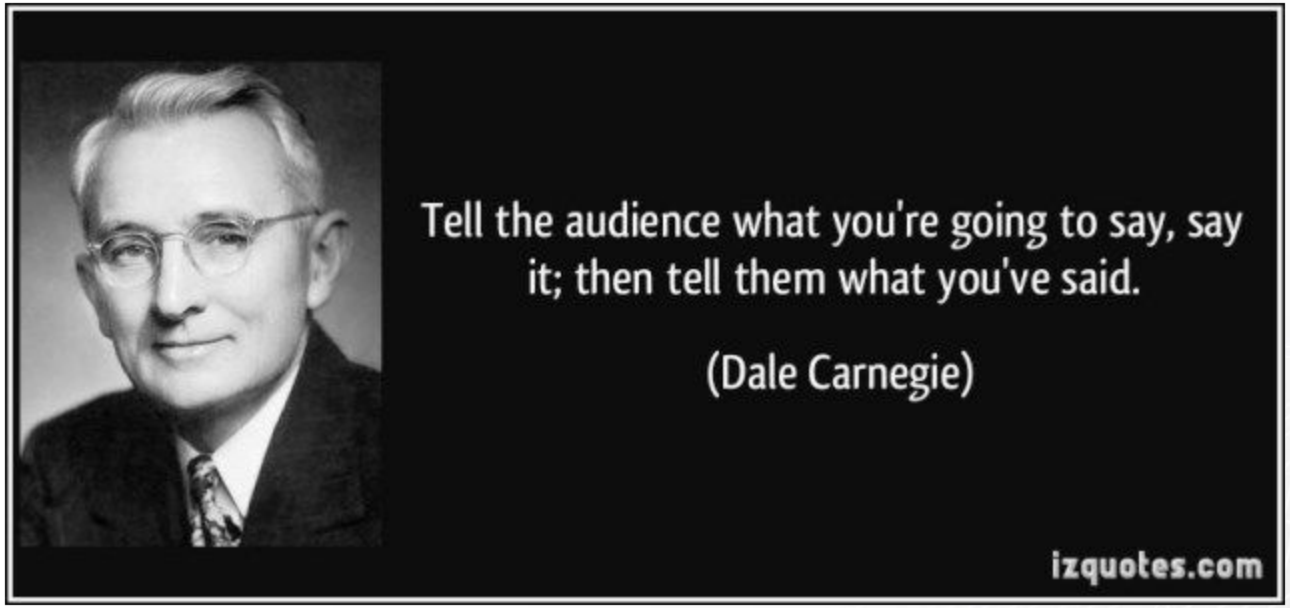
Macro 2.0 feature is in progress- a second kind of declarative macro

that will work in a similar fashion but fix some of these edge cases.

After that update, `macro_rules!` will be effectively deprecated.

redis
ENGINEERING

Summary


Tell the audience what you're going to say, say it; then tell them what you've said.
(Dale Carnegie)
izquotes.com

Thank you

Yael Tzirulnikov

redis ENGINEERING

Rust Const Generics

Gil Dafnai, Software Engineer,
Redis

redis
ENGINEERING

Introductions

Gil Dafnai

Software Engineer

Agenda

1. Motivation and Intuition
2. General Information
3. Nightly Features
4. References and other sources

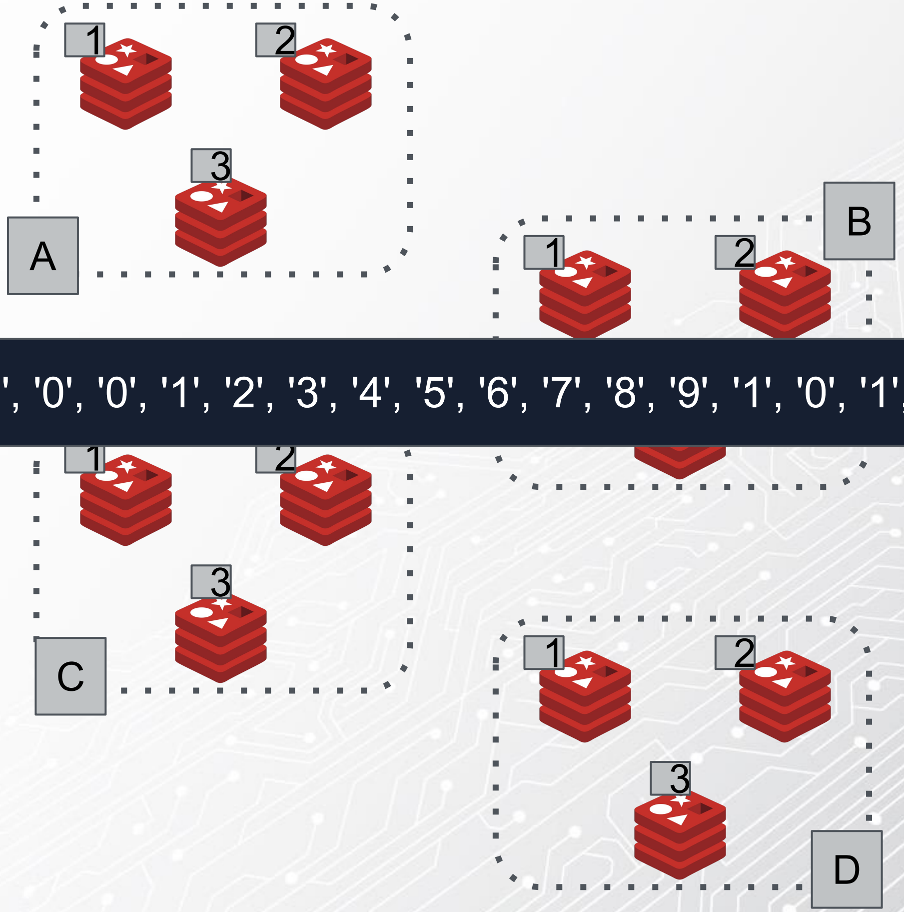
redis ENGINEERING

A Real Life (?) Example

```
#[derive(Debug)]
struct ClusterID {
    value: [char; 32],
}
```

'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '1', '0', '1', '1', '1', '2']

```
struct NodeID{
    value: [char; 40],
}
```



Implementing ID

One implementation for each struct

```
struct ClusterID {
    value: [char; 32],
}

impl Debug for ClusterID {
    ...
}
```

```
struct NodeID{
    value: [char; 40],
}

impl Debug for NodeID {
    ...
}
```

redis ENGINEERING

Implementing ID

Parameterize the Length

- We are just Moving the problem to run time.

```
struct ID(Vec<char>, usize)
```

```
struct ClusterID {
    value: ID,
}
impl ClusterID {
    const LENGTH: usize = 32
}

struct NodeID {
    value: ID,
}
impl NodeID {
    const LENGTH: usize = 40
}
```

if you have the time go read
Why Static Languages Suffer From Complexity
Great comparison of applying static and dynamic patterns to the same problem

redis ENGINEERING

Implementing ID

Parameterize the Length

- Vector allow us to parameterized the Length - but at run time
- We want it at compile time!
- Just like ***Vec<String> != Vec<u32>*** we should be able to say to ***ID<8> != ID<10>*** (but both are still IDs)

Implementing ID

Use Const Generics

- An ID can still be an array of any length. But it has to be a specific one.
- We can now implement Debug (or any trait) for any ID.

```
struct ID<const LEN: usize>([char; LEN])

struct ClusterID {
    value: ID<32>,
}

struct NodeID {
    value: ID<40>,
}
```

redis ENGINEERING

Const Generics

General Information

- ***Const generics are generic arguments that range over constant values, rather than types or lifetimes.***
- This allows, for instance, types to be parameterized by integers.
- Arrays already allow specifying their length ([T; N]) but not in a generic way (think about trying to implement a trait for an array of specific length)

redis ENGINEERING

Const Generics

General Information

- Const Generics were stabilized and released in version 1.51 and are currently in MVP
- Only Integral types are supported - Signed and Unsigned integers, char and bool.
  - So this is used mostly for generalizing arrays
- There are still important features under development

Const Generics

Another example from standard library - *array_chuncks*

- Now you can specify the chunk size. And the chunk size if part of the iterator type.
- There is also a similar function for iterator chunks (the example here is for slices)

```
pub fn array_chunks<const N: usize>(&self) -> ArrayChunks<'_, T, N> {
    ...
}
```

```
// array_chunks
let slice = ['l', 'o', 'r', 'e', 'm'];
let mut iter = slice.array_chunks();
assert_eq!(iter.next().unwrap(), &['l',
'o']);
assert_eq!(iter.next().unwrap(), &['r',
'e']);
assert!(iter.next().is_none());
assert_eq!(iter.remainder(), &['m']);
```

```
// chunks (old, not typed)
let slice = ['l', 'o', 'r', 'e', 'm'];
let mut iter = slice.chunks(2);
assert_eq!(iter.next().unwrap(), &['l',
'o']);
assert_eq!(iter.next().unwrap(), &['r',
'e']);
assert_eq!(iter.next().unwrap(), &['m']);
assert!(iter.next().is_none());
```

MinSlice<T,N>

Performance Improvements

- MinSlice is a slice with know minimal size .
- Just like slice it is unsized
- It allows the compiler to verify access to array items without runtime validation

```
pub struct MinSlice<T, const N: usize>
{

    pub head: [T; N],

    pub tail: [T],
}
```

redis
ENGINEERING

MinSlice<T,N>

Performance Implications

```
let slice: &[u8] = b"Hello, world";
let value: Option<&u8> = slice.get(6);
assert!(value.is_some());
```

- but the compiler can't know that ***value*** is a valid reference

```
let slice: &[u8] = b"Hello, world";
let minslice = MinSlice::<u8, 12>::from_slice(
                slice).unwrap();
let value: u8 = minslice.head[6];
assert_eq!(value, b' ')
```

- Length check is performed when we construct a MinSlice
- If the `unwrap()` succeeds, no more checks are needed

redis
ENGINEERING

Const Generics

Complex Expressions (Nightly Only)

```
fn foo<const N: usize>(arr: [char; N]) -> [char; N + 1] {
    ...
}
```

```
struct Grid<T, const W: usize, const H: usize> {
    array: [T; W * H],
}
```

redis ENGINEERING

Const Generics

Compile Time Validations (Nightly Only)

- The compiler will prevent runtime errors at compile time
- This specific example is event more powerful when you think about compiling for different platforms and architecture. environments, architectures, etc…

```
pub fn fill_array(number: u64, array: &mut [char]) {
    ... // write number into array and pad with zeros
}

pub fn build_array<const LENGTH: usize>(number: u64) -> [char; LENGTH]
where
    [(); LENGTH - 16]:,
{
    let mut array = ['0'; LENGTH];
    fill_array(number, &mut array);
    return array;
}
```

redis
ENGINEERING

Const Generics

More Sources and Examples

- StaticVec - fixed-capacity stack-allocated Vec.
    - also implemented a ***StaticString*** struct (***StaticVec<u8, N>***)
- Implementing Sha2 with Const Generics

```
pub struct Hash<T, const BLOCKSIZE: usize, const ROUNDS: usize> {
    pub k_constants: [T; ROUNDS], // ROUNDS = 64 or 80
    pub hash: [T; 8],
    pub scramble_funcs: ScramblePool<T>, // scrambling functions sigma etc
    pub block: [u8; BLOCKSIZE],          // BLOCKSIZE = 64 or 128
}
```

Const Generics

More Sources and Examples

```
#![feature(adt_const_params)]
#![allow(incomplete_features)]

#[derive(PartialEq, Eq)]
pub enum State {
    Init,
    Accumulate,
    Freeze,
}

pub struct Machine<const S: State> {
    total: u32
}
```

Implementing a [StateMachine]

based on *Const Generics*

```
impl Machine<{State::Init}> {
    pub fn new() -> Self {
        Self {
            total: 0
        }
    }

    pub fn accumulate(self) -> Machine<{State::Accumulate}> {
        Machine {
            total: self.total
        }
    }
}

impl Machine<{State::Accumulate}> {
    pub fn add(&mut self, add: u32) {
        self.total += add
    }

    pub fn freeze(self) -> Machine<{State::Freeze}> {
        Machine {
            total: self.total
        }
    }
}

impl Machine<{State::Freeze}> {
    pub fn unwrap(self) -> u32 {
        self.total
    }
}
```

redis
ENGINEERING

Thank you

Gil Dafnai

FFI - How C & Rust can be BFFs?

Sharon Rosenfeld/Principal Software Engineer, redis

redis
ENGINEERING

Introductions

Sharon Rosenfeld

Principal Software Engineer

redis ENGINEERING

Agenda

1. Our FFI Use Case
2. 1st phase - basic wrapping
3. 2nd phase - idiomatic wrapping
4. The callback challenge

Our use case

- We write a redis module in Rust
- We needed a Raft Client for our module
- Raft Lib is written in C :(

1st Phase - Basic Wrapping

- BindGen
- OSS tool that consumes C/C++ headers and **generates** Rust FFI bindings
  https://github.com/rust-lang/rust-bindgen

Bindgen Output

Doggo.h

```c
typedef struct Doggo {
    int many;
    char wow;
} Doggo;


void eleven_out_of_ten_majestic_af(Doggo*
pupper);
```

Doggo.rs

```rust
/* automatically generated by rust-bindgen*/
#[repr(C)]
pub struct Doggo {
  pub many: ::std::os::raw::c_int,
  pub wow: ::std::os::raw::c_char,
}
extern "C" {
    pub fn
eleven_out_of_ten_majestic_af(pupper: *mut
Doggo);
}
```

redis
ENGINEERING

2nd Phase - Idiomatic Wrapping

- Wrapper Layer
- Idiomatic standardization
  - Return type
  - Simple Arguments
  - Struct Arguments

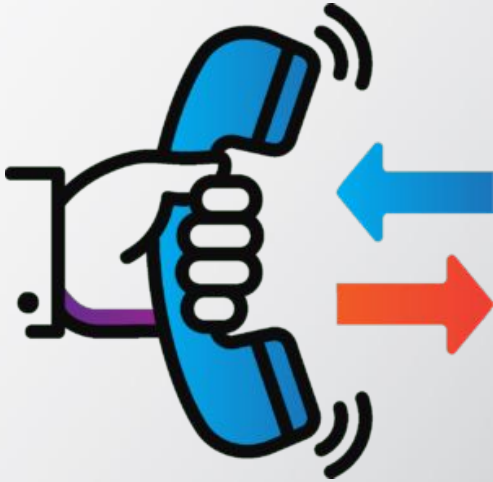
redis ENGINEERING

Wrapper Function Example

```
pub fn recv_snapshot_response(
    &self,
    id: RaftNodeId,
    resp: &RaftSnapshotResponse,
) -> Result<(), RaftError> {
    let mut raw_resp: raft_snapshot_resp_t = resp.into();
    let res = unsafe {
        bindings::raft_recv_snapshot_response(
            id,
            &mut raw_resp as *mut raft_snapshot_resp_t,
        )
    };
```

The Callback Challenge

```
extern "C" {
    pub fn raft_recv_read_request(
        cb: raft_read_request_callback_f,
        cb_arg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
```



```
pub type raft_read_request_callback_f = ::std::option::Option<
    unsafe extern "C" fn(arg: *mut ::std::os::raw::c_void, can_read:
::std::os::raw::c_int),
>;
```

Callback Flow

Application

passes a boxed CB struct

Wrapper

passes predefined CB + the “CB object” from app.

Binding

C Library

Wrapper callback

Unfold the opaque context and call application callback

Application callback

Wrapper Function

```
pub fn recv_read_request<C>(&self, read_ctx: Box<C>) -> Result<(), RaftError>
Where       C: ReadCBCtx,
{
      let ctx_box_ptr = Box::into_raw(read_ctx);
      let ctx_ptr = unsafe { transmute(ctx_box_ptr) };
      let res = unsafe {
         bindings::raft_recv_read_request(
              self.inner,
              Some(callbacks::read_request_callback::<C>),
              ctx_ptr,
         )
    };
```

coldfusion

Wrapper Callback Code

```
pub unsafe extern "C" fn read_request_callback<C>(arg: *mut c_void, can_read:
c_int)
Where C: ReadCBCtx,
{
    let ptr = arg as *mut C;
    let ctx = Box::from_raw(ptr);
    if !can_read {
      ctx.read(Err(EzError::ReadTimeout));
     } else {
      ctx.read(Ok(self.state));
     }
```

redis ENGINEERING

To Wrap Things Up …

- Bindgen
- Turning binding code to beautiful rust code
- Passing objects between rust , c and rust

Thank you

Sharon Rosenfeld

redis ENGINEERING