rkyv (*archive*) is a zero-copy deserialization framewo

This book covers the motivation, architecture, and n way to learn and understand rkyv, but won't go as ir documentation will. Don't be afraid to consult these you read through.

Resources

Learning Materials

- The rkyv discord is a great place to get help wi people using rkyv
- The rkyv github hosts the source and tracks pr

Documentation

- rkyv, the core library
- rkyv_dyn, which adds trait object support to rk
- rkyv_typename, a type naming library

Benchmarks

- The rust serialization benchmark is a shootout rust serialization solutions. It includes special b solutions like rkyv.

Sister Crates

- bytecheck, which rkyv uses for validation

- ptr_meta, which rkyv uses for pointer manipul
- rend, which rkyv uses for endian-agnostic featu

Motivation

First and foremost, the motivation behind rkyv is im
achieves that goal can also lead to gains in memory
the way.

> Familiarity with other serialization frameworks ar
> works will help, but isn't necessary to understanc

Most serialization frameworks like serde define an i
basic types such as primitives, strings, and byte arra
type into two stages: the frontend and the backend.
breaks it down into the serializable types of the data
data model types and writes them using some data
etc. This allows a clean separation between the seri
it is written to.

> Serde describes its data model in the serde book
> eventually boils down to some combination of th

A major downside of traditional serialization is that
time to read, parse, and reconstruct types from thei

> In JSON for example, strings are encoded by surr
> quotes and escaping invalid characters inside of t
>
> ```
> { "line": "\"All's well that ends well\""
>           ^^                            ^ ^
> ```
>
> numbers are turned into characters:
>
> ```
> { "pi": 3.1415926 }
>         ^^^^^^^^^
> ```
>
> and even field names, which could be *implicit* in r
>
> ```
> { "message_size": 334 }
>   ^^^^^^^^^^^^^^^
> ```
>
> All those characters are not only taking up space,

time we read and parse JSON, we're picking throu
figure out what the values are and reproduce the
bytes of memory, but it's encoded using nine byt
nine characters into the right `f32`!

This deserialization time adds up quickly, and in dat
and media editing it can come to dominate load tim
serialization technique called *zero-copy deserializatio*

Zero-copy deserializatio

Zero-copy deserialization is a technique that reduce
access and use data by *directly referencing bytes in th*

> This takes advantage of how we have to have sor
> to deserialize it. If we had some JSON:
>
> ```
> { "quote": "I don't know, I didn't listen.
>            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
> ```
>
> Instead of copying those characters into a `Strin`
> JSON buffer as a `&str` . The lifetime of that `&str`
> we wouldn't be allowed to drop it until we had dr

Partial zero-copy

Serde and others have support for partial zero-copy
pieces of the deserialized data are borrowed from t
example, can borrow their bytes directly from the s
bincode that don't perform any character escaping.
created to hold the deserialized length and point to

> A good way to think about this is that even thoug
> the buffer, we still have to parse the *structure* out
>
> ```
> struct Example<'a> {
>   quote: &'a str,
>   a: &'a [u8; 12],
>   b: u64,
>   c: char,
> }
> ```
>
> So a buffer might break down like this:
>
> ```
> I don't know, I didn't listen.AAAAAAAAAAAA
> ^-----------------------------^-----------
>  quote: str                    a: [u8; 12]
> ```
>
> We do a lot less work, but we still have to parse, c

```
Example {
  quote: str::from_utf8(&buffer[0..30]).un
  a: &buffer[30..42],
  b: u64::from_le_bytes(&buffer[42..50]),
  c: char::from_u32(u32::from_le_bytes(&bu
}
```

And we can't borrow types like `u64` or `char` that
since our buffer might not be properly aligned. W
store those! Even though we borrowed 42 of the
the last 12 and still had to parse through the buff

Partial zero-copy deserialization can considerably in
speed up some deserialiation, but with some work w

Total zero-copy

rkyv implements total zero-copy deserialization, whi
during deserialization and no work is done to deser
structuring its encoded representation so that it is t
representation of the source type.

This is more like if our buffer *was* an Example:

```
struct Example {
  quote: String,
  a: [u8; 12],
  b: u64,
  c: char,
}
```

And our buffer looked like this:

```
I don't know, I didn't listen.__QOFFQLENAA
^-----------------------------  ^---^---^-
 quote bytes                    pointer  a
                                and len
                                ^---------
                                 Example
```

In this case, the bytes are padded to the correct a
`Example` are laid out exactly the same as they wo
deserialization code can be much simpler:

```
unsafe { &*buffer.as_ptr().add(32).cast()
```

This operation is almost zero work, and more imp
data. No matter how much or how little data we h
offset and a cast to access our data.

This opens up blazingly-fast data loading and enable
more quickly than traditional serialization.

Architecture

The core of rkyv is built around relative pointers and `Serialize`, and `Deserialize`. Each of these traits h supports unsized types: `ArchiveUnsized`, `Serializ`

> A good way to think about it is that sized types ar are built on. That's not a fluke either, rkyv is built more complex abstractions out of lower-level ma way. It's not much different from what you norm

---

The system is built to be flexible and can be extende example, the `rkyv_dyn` crate adds support for trait defining how they build up to allow trait objects to b

Relative pointers

Relative pointers are the bread and butter of total z
replacing the use of normal pointers. But why can't

Consider some zero-copy data on disc. Before we ca
memory. But we can't control *where* in memory it ge
could be located at a different address, and therefo
located at a different address.

> One of the major reasons for this is actually *secu*
> program, it may run in a completely different ran
> called address space layout randomization and it
> memory corruption vulnerabilities.
>
> At most, we can only control the *alignment* of our
> work within those constraints.

This means that we can't store any pointers to that
soon as we reload the data, it might not be at the sa
pointers dangling, and would almost definitely resul
other libraries like abomonation store some extra d
takes the place of deserialization, but we can do bet

> In order to perform that fixup step, abomonation
> *mutable backing*. This is okay for many use cases,
> won't be able to mutate our buffer. One example
> files.

While normal pointers hold an absolute address in
offset to an address. This changes how the pointer

Pointer	Self is moved	
Absolute	✅ Target is still at address	❌
Relative	❌ Relative distance has changed	✅ di

This is exactly the property we need to build data st
deserialization. By using relative pointers, we can lo
and still have valid pointers inside of it. Relative poi
memory either, so we can memory map entire files

data in a structured manner.

rkyv's implementation of relative pointers is the `Rel`

Archive

Types that implement `Archive` have an alternate re
deserialization. The construction of archived types h

1. Any dependencies of the type are serialized. Fo
   of the string, for boxes it would be the boxed v
   contained elements. Any bookkeeping from th
   type and held onto for later. This is the *serializ*
2. The resolver and original value are used to cor
   output buffer. For strings the resolver would b
   boxes it would be the position of the boxed va
   position of the archived elements. With the ori
   the archived version can be constructed. This i

Resolvers

A good example of why resolvers are necessary is w
two strings:

```
let value = ("hello".to_string(), "world".to_
```

The archived tuple needs to have both of the strings

```
0x0000      AA AA AA AA BB BB BB BB
0x0008      CC CC CC CC DD DD DD DD
```

A and B might be the length and pointer for the first
might be the length and pointer for the second strir

When archiving, we might be tempted to serialize ar
serialize and resolve the second one. But this might
("world") between the two! Instead, we need to write
then finish archiving both of them. The tuple doesn'
need to finish archiving themselves, so they have to
Resolver.

This way, the tuple can:

1. Archive the first string (save the resolver)
2. Archive the second string (save the resolver)
3. Resolve the first string with its resolver

4. Resolve the second string with its resolver

And we're guaranteed that the two strings are place
need.

Serialize

Types implement `Serialize` separately from `Arch` some object, then `Archive` turns the value and tha Having a separate `Serialize` trait is necessary bec one archived representation, you may have options order to create one.

> The `Serialize` trait is parameterized over the *se* mutable object that helps the type serialize itself. `char` don't *bound* their serializer type because th any kind of serializer. More complex types like `Bo` that implements `Serializer`, and even more co require a serializer that additionally implement `S` `ScratchSpace`.

Unlike `Serialize`, `Archive` doesn't parameterize c shouldn't matter what serializer a resolver was mad

Serializer

rkyv provides serializers that provide all the functio library types, as well as serializers that combine oth all of the components' capabilities.

The provided serializers offer a wide range of strate cases will be best suited by `AllocSerializer`.

> Many types require *scratch space* to serialize. This they can use temporarily and return when they'r request scratch space to store the resolvers for it of them. Requesting scratch space from the seria reused many times, which reduces the number o performed while serializing.

Deserialize

Similarly to `Serialize`, `Deserialize` parameterize
converts a type from its archived form back to its or
deserialization occurs in a single step and doesn't h

> `Deserialize` also parameterizes over the type th
> allows the same archived type to deserialize into
> depending on what's being asked for. This helps
> abstractions, but might require you to annotate t

This provides a more or less a traditional deserializa
sped up somewhat by having very compatible repre
memory and performance penalties of traditional d
what you need before you use it. Deserialization is r
as long as you can do so through the archived versi

> Even the highest-performance serialization frame
> speed limit because of the amount of memory all
> performed.

A good use for `Deserialize` is deserializing portion
the archived data to locate some subobject, then de
archive as a whole. This granular approach provides
deserialization as well as traditional deserialization.

Deserializer

Deserializers, like serializers, provide capabilities to
types don't bound their deserializers, but some like
order to deserialize memory properly.

Alignment

The *alignment* of a type restricts where it can be loca
loads and stores. Because rkyv creates references t
bytes, it has to ensure that the references it creates

> In order to perform arithmetic and logical operat
> *load* that data from memory into its registers. Ho
> limitation on how the CPU can access that data: it
> *word boundaries*. These words are the natural siz
> word size is 4 bytes for 32-bit machines and 8 byt
> had some data laid out like this:
>
> ```
> 0   4   8   C
> AAAABBBBCCCCDDDD
> ```
>
> On a 32-bit CPU, accesses could occur at any add
> example, one could access `A` by loading 4 bytes
> bytes from address 4, and so on. This works grea
> word boundaries. *Unaligned* data can throw a wr
>
> ```
> 0   4   8   C
> ..AAAABBBBCCCC
> ```
>
> Now if we want to load `A` into memory, we have
>
> 1. Load 4 bytes from address 0
> 2. Throw away the first two bytes
> 3. Load 4 bytes from address 4
> 4. Throw away the last two bytes
> 5. Combine our four bytes together
>
> That forces us to do twice as many loads *and* per
> can have a real impact on our performance acros
> our data to be properly aligned.

rkyv provides two main utilities for aligning byte buf

- `AlignedVec` is a drop-in replacement for `Vec<`
- `AlignedBytes` is a wrapper around `[u8; N]`

Both of these types align the bytes inside to 16-byte
for almost all use cases, but if your particular situati

then you may need to manually align your bytes.

In practice

rkyv has a very basic unaligned data check built in th
also validate your data, then it will always make sure

Common pitfalls

In some cases, your archived data may be prefixed b
the buffer. If this extra data misaligns the following
the prefixing data removed before accessing it.

In other cases, your archived data may not be tight t
`archived_root` rely on the end of the buffer being t
miscalculate the positions of the contained values if

Format

Types which derive `Archive` generate an archived v

- Member types are replaced with their archivec
- Enums have `#[repr(N)]` where N is `u8` , `u16` ,
  smallest possible type that can represent all of

For example, a struct like:

```
struct Example {
    a: u32,
    b: String,
    c: Box<(u32, String)>,
}
```

Would have the archived counterpart:

```
struct ArchivedExample {
    a: u32,
    b: ArchivedString,
    c: ArchivedBox<(u32, ArchivedString)>,
}
```

With the `strict` feature, these structs are addition
guaranteed portability and stability.

> In most cases, the `strict` feature will not be nec
> efficiency of archived types. Make sure you under
> read the crate documentation for details on the

---

rkyv provides `Archive` implementations for commo
general they follow the same format as derived imp
cases. For example, `ArchivedString` performs a sm
reduce memory use.

Object order

rkyv lays out subobjects in depth-first order from th
the root object is stored at the end of the buffer, no
tree:

```
   a
  / \
 b   c
    / \
   d   e
```

would be laid out like this in the buffer:

```
b d e c a
```

from this serialization order:

```
a -> b
a -> c -> d
a -> c -> e
a -> c
a
```

This deterministic layout means that you don't neec object in most cases. As long as your buffer ends rig can use `archived_root` with your buffer.

Wrapper types

Wrapper types make it easy to customize the way th
make it easier to adapt rkyv to existing data models
deserializing idiomatic for even complicated types.

Annotating a field with `#[with(...)]` will *wrap* that
struct is serialized or deserialized. There's no perfor
but doing more or less work during serialization anc
performance. This excerpt is from the documentatic

```
#[derive(Archive, Deserialize, Serialize)]
struct Example {
    #[with(Incremented)]
    a: i32,
    // Another i32 field, but not incrementec
    b: i32,
}
```

The `Incremented` wrapper is wrapping `a`, and the c
incremented in its archived form.

`With`

The core type behind wrappers is `With`. This struct
another name for the type inside of it. rkyv uses `Wi`
serializing and deserializing, and when you write you
with `With` as well.

See `ArchiveWith` for an example of how to write yc

Shared Pointers

The implementation details of shared pointers may
Specifically, the rules surrounding how and when sh
and pooled may affect how you choose to use them

Serialization

Shared pointers (`Rc` and `Arc`) are serialized whene
time, and the data address is reused when subsequ
data. This means that you can expect shared pointe
when archived, even if they are unsized to different

Weak pointers (`rc::Weak` and `sync::Weak`) have se
they're encountered. The serialization process upgr
serializes them like shared pointers. Otherwise, it se

Deserialization

Similarly, shared pointers are deserialized on the fir
Weak pointers do a similar upgrade attempt when t

Serializers and Deserializers

The serializers for shared pointers hold the location
safe to serialize shared pointers to an archive acros
you use the same serializer for each one. Using a ne
but may end up duplicating the shared data.

The deserializers for shared pointers hold a shared
and will hold them in memory until the deserializer
serialize only weak pointers to some shared data, th
when deserialized but will point to nothing as soon

Unsized Types

rkyv supports unsized types out of the box and ship
common unsized types (`str` s and slices). Trait obje
`rkyv_dyn` , see "Trait Objects" for more details.

Metadata

The core concept that enables unsized types is meta
different sizes, in contrast with languages like C and
size. This is important for the concept of sizing, whic
rust's Sized trait.

Pointers are composed of two pieces: a data addres
address is what most people think of when they thir
the pointed data. The metadata for a pointer is som
safely with the data at the pointed location. It can be
`Sized` types. Pointers with no extra metadata are s
pointers *with* metadata are sometimes called "wide'

> rkyv uses the `ptr_meta` crate to perform these c
> these may be incorporated as part of the standar

Fundamentally, the metadata of a pointer exists to p
information to safely access, drop, and deallocate st
slices, the metadata carries the length of the slice, fo
function table (vtable) pointer, and for custom unsiz
the single trailing unsized member.

Archived Metadata

For unsized types, the metadata for a type is archive
to the data. This mirrors how rust works internally t
and other exotic use cases. This does complicate thi
the metadata archiving process will end up as just fi
returning `()` .

This is definitely one of the more complicated pa
difficult to wrap your head around. Reading the c
may help you understand how the system works

Trait Objects

Trait object serialization is supported through the `r` maintained as part of rkyv, but is separate from the implementations to be used instead. This section wi of `rkyv_dyn` and how to use it effectively.

> `rkyv_dyn` may not work in some exotic environm to register trait objects. If you want these capabili your environment, feel free to file an issue or dro through.

Core traits

The new traits introduced by `rkyv_dyn` are `Serial` are effectively type-erased versions of `SerializeUn` that the traits are object-safe. Likewise, it introduces and deserializers: `DynSerializer` and `DynDeserial` basic functionality required to serialize most types, l custom types require.

> `DynSerializer` implements the `Serializer` and not be suitable for all use cases. If you need more by in the discord to talk it through.

Architecture

It is highly recommended to use the provided `arch` traits and set everything up correctly.

Using `archive_dyn` on a trait definition creates ano your trait and `SerializeDyn`. This "shim" trait is bla implement your trait and `SerializeDyn`, so you sho trait to use it.

The shim trait should be used everywhere that you you want to serialize. By default, it will be named "S

approach that similar libraries take is directly adding
your trait. While more ergonomic, this approach do
the trait on types that cannot or should not implem
approach was favored for `rkyv_dyn`.

When the shim trait is serialized, it stores the type h
metadata so it can get the correct vtable for it when
vtables for implementing types must be known ahe
`archive_dyn` for the second time.

Using `archive_dyn` on a trait implementation regis
implementation with a global lookup, allowing it to b
process can be slow, the `vtable_cache` feature allo
only the first time, then cached locally for future loo
alternate implementations may take a different app
benefits and tradeoffs.

Validation

Validation can be enabled with the `validation` feat
`bytecheck` crate to perform archive validation, and
and malicious data.

To validate an archive, you first have to derive `Chec`

```
use rkyv::{Archive, Deserialize, Serialize};

#[derive(Archive, Deserialize, Serialize)]
#[archive(check_bytes)]
pub struct Example {
    a: i32,
    b: String,
    c: Vec<bool>,
}
```

The `#[archive(check_bytes)]` attribute derives `Ch`
Finally, you can use `check_archived_root` to check
archived value if it was successful:

```
use rkyv::check_archived_root;

let archived_example = check_archived_root::<
```

More examples of how to enable and perform valid
crate's `validation` module.

The validation context

When checking an archive, a validation context is cr
defaults that will work for most archived types. If yo
logic, you may need to augment the capabilities of t
check your type and use `check_archived_root_wit`

> The `DefaultValidator` supports all builtin rkyv t
> whether you have the `alloc` feature enabled or

Bounds checking and subtree ra

All pointers are checked to make sure that they:

- point inside the archive
- are properly aligned
- and have enough space afterward to hold the

However, this alone is not enough to secure against sharing violations, so rkyv uses a system to verify th ownership model.

Archive validation uses a memory model where all s memory. This is called a *subtree range*. When validat keeps track of where subobjects are allowed to be l range from the beginning with `push_prefix_subtre` `push_suffix_subtree_range`. After pushing a subtr can be checked by calling their `CheckBytes` implem checked, `pop_prefix_subtree_range` and `pop_suf` restore the original range with the checked section r

Validation and Shared Pointers

While validating shared pointers is supported, some prevent malicious data from validating:

Shared pointers that point to the same object will fa types. This can cause issues if you have a shared po pointers are an array pointer and a slice pointer. Sir shared pointers to the same value as a concrete typ `dyn Any`).

rkyv still supports these use cases, but it's not possi with these use cases. Alternative validation solution hashes may be a better approach in these cases.

Feature Comparison

This is a best-effort feaure comparison between rky
by no means completely comprehensive, and pull re
welcomed.

Feature matrix

Feature	rkyv
Open type system	yes
Scalars	yes
Tables	no*
Schema evolution	no*
Zero-copy	yes
Random-access reads	yes
Validation	upfront*
Reflection	no*
Object order	bottom-up
Schema language	derive
Usable as mutable state	yes
Padding takes space on wire?	yes*
Unset fields take space on wire?	yes
Pointers take space on wire?	yes
Cross-language	no
Hash maps and B-trees	yes
Shared pointers	yes

* *rkyv's open type system allows extension types that p*

Open type system

One of rkyv's primary features is that its type systen
write custom types and control their properties very
solid foundation to build many other features on to
already a fundamental part of how rkyv works.

Unsized types

Even though they're part of the main library, unsize serialization functionality. Types like `Box` and `Rc/A` entry points for unsized types into the sized system

Trait objects

Trait objects are further built on top of unsized type objects easy and safe.

FAQ

Because it's so different from traditional serializatio
questions about rkyv. This is meant to serve as a co
answers.

How is rkyv zero-copy? It definit
into memory.

Traditional serialization works in two steps:

1. Read the data from disk into a buffer (maybe i
2. Process the data in the buffer into the deserial

The copy happens when the data in the buffer ends
Zero-copy deserialization doesn't deserialize the bu
avoids this copy.

You can actually even avoid reading the data from d
environments by using memory mapping.

How does rkyv handle endianne

rkyv supports three endiannesses: native, little, and
little or big, but removes the abstraction layer to mo
types.

You can enable specific endiannesses with the `litt`

Is rkyv cross-platform?

Yes, but rkyv has been tested mostly on x86 machin
need to get fixed for other architectures.

Can I use this in embedded and
environments?

Yes, disable the `std` feature for `no_std`. You can a
to disable all memory allocation capabilities.

Safety

Isn't this very unsafe if you acce

Yes, *but* you can still access untrusted data if you va
It's an extra step, but it's usually still less than the co
format. rkyv has proven to round-trip faster than bi

Doesn't that mean I always have

No. There are many other ways you can verify your
and signed buffers.

Isn't it kind of deceptive to say r require validation?

The fastest path to access archived data is marked a
unusable, it means that it's only safe to call if you ca

> The value must be archived at the given position

As long as you can (reasonably) guarantee that, the
every archive needs to be validated, and you can us
guarantee data integrity and security.

Even if you do need to always validate your data be
faster than deserializing with other high-performan
even though it's not by the same margins.

Contributors

Thanks to all the contributors who have helped doc

- David Koloski (djkoloski)

If you feel you're missing from this list, feel free to a