A (relatively) new datetime library for Rust

https://github.com/BurntSushi/jiff

2026-01-22

Andrew Gallant (aka BurntSushi)

About Me

- Part of the Rust project for 10 years.
  - Currently on the `libs-api` and `regex` teams.
  - Primarily work on ecosystem libraries: `regex`, `memchr`, `bstr`, `csv`, `fst`, `ignore`, `walkdir`.
  - Wrote ripgrep
- Work at Astral on improving Python tooling using Rust.

What is a datetime library?

Program:

```
fn main() {
    let now = std::time::SystemTime::now();
    println!("{now:?}");
}
```

Output:

```
SystemTime { tv_sec: 1768932362, tv_nsec: 181638717 }
```

What is a datetime library?

Program:

```
fn main() {
    let now = jiff::Zoned::now();
    println!("{now:?}");
}
```

Output:

```
2026-01-20T13:14:32.102211075-05:00[America/New_York]
```

Why build another datetime library?

- Hold that thought

Example: What day is my birthday in 2030?

Program:

```
fn main() {
    let birthday = const { jiff::civil::date(2030, 7, 5) };
    println!("{:?}", birthday.weekday());
}
```

Output:

```
Friday
```

Example: Next five birthdays?

```
use jiff::ToSpan;

fn main() -> anyhow::Result<()> {
    let now = jiff::Zoned::now();
    let birthday = now.with().month(7).day(5).build()?;
    for zdt in birthday.series(1.year()).filter(|x| x >= now).take(5) {
        println!("{} {:?}", zdt.date(), zdt.weekday());
    }
    Ok(())
}
```

```
2026-07-05 Sunday
2027-07-05 Monday
2028-07-05 Wednesday
2029-07-05 Thursday
2030-07-05 Friday
```

What is a `Zoned`?

```
use jiff::{tz::TimeZone, Timestamp, Zoned};

fn main() -> anyhow::Result<()> {
    let timestamp = Timestamp::UNIX_EPOCH;
    let zdt = Zoned::new(timestamp, TimeZone::UTC);
    println!("{zdt}");
    let zdt = zdt.in_tz("America/New_York")?;
    println!("{zdt}");
    let zdt = zdt.in_tz("Australia/Sydney")?;
    println!("{zdt}");

    Ok(())
}
```

```
1970-01-01T00:00:00+00:00[UTC]
1969-12-31T19:00:00-05:00[America/New_York]
1970-01-01T10:00:00+10:00[Australia/Sydney]
```

Example: How old am I?

```
use jiff::civil;

fn main() -> anyhow::Result<()> {
    let dob = const { civil::date(1987, 7, 5) }.in_tz("America/New_York")?;
    let now = jiff::Zoned::now();
    let age = now.since(&dob)?;
    println!("{age:#}");
    Ok(())
}
```

```
337911h 11m 9s 105ms 160µs 972ns
```

Example: How old am I? (redux)

```
use jiff::civil;

fn main() -> anyhow::Result<()> {
    let dob = const { civil::date(1987, 7, 5) }.in_tz("America/New_York")?;
    let now = jiff::Zoned::now();
    let age = now.since((jiff::Unit::Year, &dob))?;
    println!("{age:#}");
    Ok(())
}
```

```
38y 6mo 15d 14h 25m 47s 242ms 151µs 816ns
```

Durations

- The main duration type in Jiff is a `Span`.
- Combines *calendar* and *time* components into one thing.
- Stores values for every individual unit: years, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds and nanoseconds.
- `2 hours` and `120 minutes` are distinct values in memory.
- Very different from `std::time::Duration`.
- (But Jiff works with `std::time::Duration` too.)

Why are durations limited to hours by default?

Using largest units isn't reversible. For example:

- `let zdt1: Zoned = "2024-03-02[America/New_York]".parse()?;`
- `let zdt2: Zoned = "2024-05-01[America/New_York]".parse()?;`
- `let span = zdt1.until((Unit::Year, &zdt2))?;`
- `"1 month 29 days"`
- `let zdt = zdt2 - span;`
- `"2024-03-03[America/New_York]"`

Limiting durations to hours provides reversibility

- `let zdt1: Zoned = "2024-03-02[America/New_York]".parse()?;`
- `let zdt2: Zoned = "2024-05-01[America/New_York]".parse()?;`
- `let span = zdt1.until(&zdt2)?;`
- `"1439 hours"`
- `let zdt = zdt2 - span;`
- `"2024-03-02[America/New_York]"`

Durations are aware of daylight saving time

```
use jiff::{ToSpan, Zoned};
fn main() -> anyhow::Result<()> {
    // `2025-11-02` was not 24 hours long in New England
    let zdt1: Zoned = "2025-11-01T17:30[America/New_York]".parse()?;

    let zdt2 = &zdt1 + 1.day();
    println!("{zdt2}");

    let zdt3 = &zdt1 + 24.hours();
    println!("{zdt3}");

    Ok(())
}
```

```
2025-11-02T17:30:00-05:00[America/New_York]
2025-11-02T16:30:00-05:00[America/New_York]
```

What is a time zone?

- A geographic location where humans agree on what their clocks say
- For programmers, it's a function
- Unambiguously maps an instant to a civil datetime
- Ambiguously maps a civil datetime to an instant

Instant -> Civil DateTime

This is the "easy" case, since the mapping is always unambiguous:

```
fn main() -> anyhow::Result<()> {
    let now = jiff::Timestamp::now();
    let zdt = now.in_tz("America/New_York")?;
    println!("{zdt:?}");

    Ok(())
}
```

```
2026-01-20T13:14:32.102211075-05:00[America/New_York]
```

Civil DateTime -> Instant

Jiff will automatically choose an instant when a mapping doesn't exist:

```
use jiff::civil;

fn main() -> anyhow::Result<()> {
    let datetime = civil::date(2026, 3, 8).at(2, 30, 0, 0);
    let zdt = datetime.in_tz("America/New_York")?;
    println!("{}", zdt.strftime("%Y-%m-%d %H:%M%P"));
    Ok(())
}
```

```
2026-03-08 03:30am
```

Civil DateTime -> Instant

And the same for when there are multiple possible choices:

```
use jiff::civil;

fn main() -> anyhow::Result<()> {
    let datetime = civil::date(2026, 11, 1).at(1, 30, 0, 0);
    let zdt = datetime.in_tz("America/New_York")?;
    println!("{}", zdt.strftime("%Y-%m-%d %H:%M%P %:z"));
    Ok(())
}
```

```
2026-11-01 01:30am -04:00
```

Civil DateTime -> Instant

Lower level APIs exist for making a different choice:

```
use jiff::civil;

fn main() -> anyhow::Result<()> {
    let tz = jiff::tz::TimeZone::system();
    let datetime = civil::date(2026, 11, 1).at(1, 30, 0, 0);
    let zdt = tz.to_ambiguous_zoned(datetime).later()?;
    println!("{}", zdt.strftime("%Y-%m-%d %H:%M%P %:z"));
    Ok(())
}
```

```
2026-11-01 01:30am -05:00
```

RFC 9557

- Relatively new, published in 2024
- Extends RFC 3339
- `2025-01-22T19:00-05:00[America/New_York]` is valid
- `2025-01-22T19:00-04:00[America/New_York]` is invalid (Jiff parse error)
- Losslessly roundtrip zoned datetimes
- Spotty support: Temporal, `java.time`, `whenever` (Python)

Wrapping up: Jiff's API design rationale

- Guides you away from making mistakes
  - e.g., Assuming UTC is explicit
  - e.g., Fixed offset datetimes are discouraged (but not impossible)
  - DST is handled for you automatically whenever possible
- Clearly distinguishes between zone-aware and not zone-aware
- Is pragmatic: provides lots of interoperable points and a standard "Unix timestamp" type
- Heavily inspired by Temporal (ECMAScript TC39 proposal, nearly stage 4)

Wrapping up: Other features

- Rounding durations (time zone aware)
- Comprehensive `strftime` and `strptime` support (used by uutils now)
- no-std and no-alloc support
- ISO 8601, RFC 3339, RCC 9557, RFC 2822, RFC 9110, RFC 9636

Wrapping up: Not Supported

- Leap seconds
- Localization
- Non-Gregorian calendars

Why build another datetime library?

- Existing popular datetime libraries: `chrono` and `time`
- API in a stalled "local maximum" (particularly `chrono`)
- Undue focus on "fixed offset" datetimes
- Spotty support for IANA Time Zone Database
- Spotty support for calendar durations
- Spotty support for daylight saving time

Links

Repository: https://github.com/BurntSushi/jiff

API docs: https://docs.rs/jiff

CLI tool: https://github.com/BurntSushi/biff