Introducing Rust-Prometheus

How Rust makes metrics ***safe*** and ***fast***

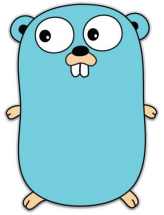
Wish Shi @ PingCAP TiDB

FOSDEM[19]

I’m...

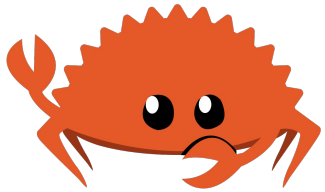
Wish, Infrastructure Engineer from PingCAP

In PingCAP...



We build distributed SQL database TiDB

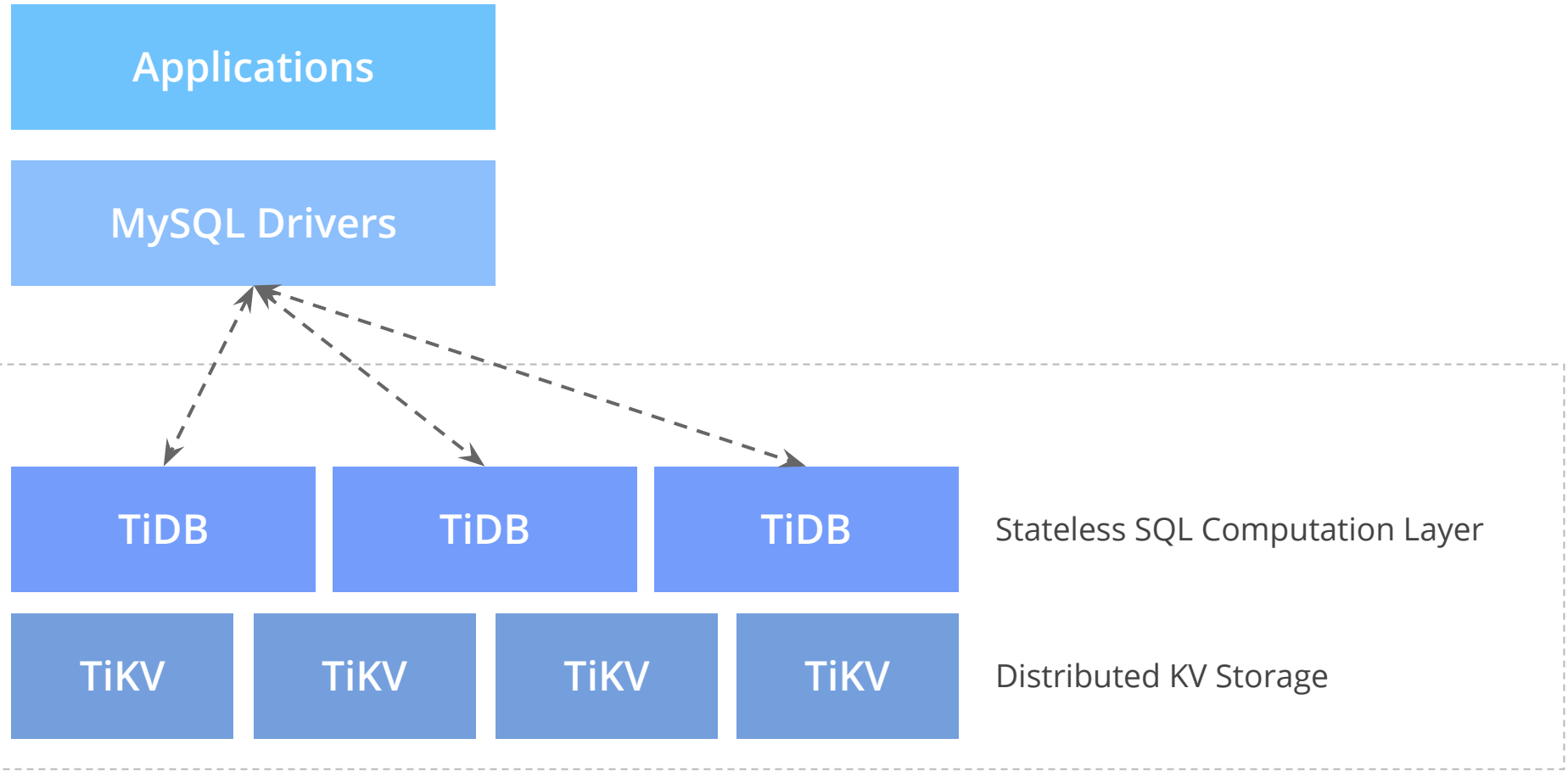
.. which is built upon the distributed KV database TiKV



TiDB & TiKV...

- 15+ PB
- 300+ customers worldwide
- Bank, Internet, Enterprise companies

The architecture of PingCAP product



In PingCAP, we also...

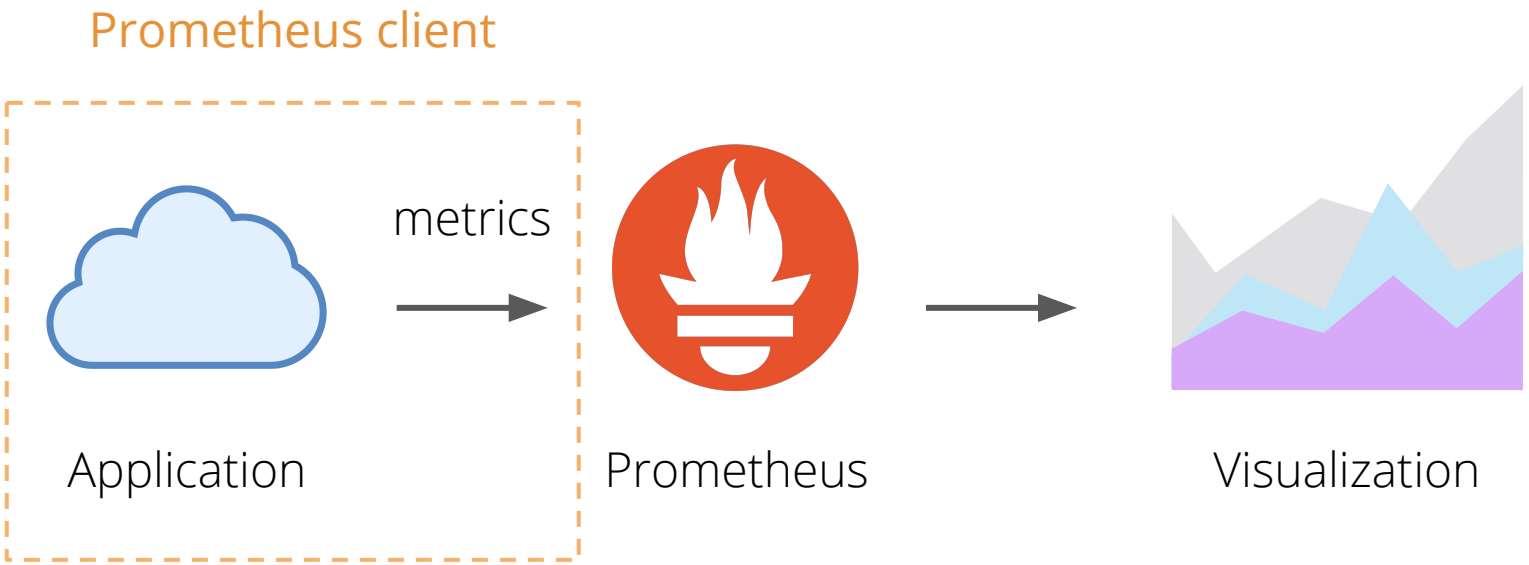
Created and maintain many Rust crates, e.g.

- rust-prometheus - Prometheus instrumentation library
- rust-rocksdb - RocksDB binding and wrapper
- raft-rs - Raft distributed consensus algorithm
- grpc-rs - gRPC library over gRPC C core and Futures
- fail-rs - Fail points

Prometheus is...

Prometheus is...

An open-source system monitoring and alerting toolkit



Rust-prometheus

It’s not a Prometheus database implemented in Rust.

Sorry for that :)

PingCAP

Getting Started

1. Define metrics

```
lazy_static! {
    static ref REQUEST_DURATION: HistogramVec = register_histogram_vec!(
        "http_requests_duration",
        "Histogram of HTTP request duration in seconds",
        &["method"],
        exponential_buckets(0.005, 2.0, 20).unwrap()
    ).unwrap();
}
```

2. Record metrics

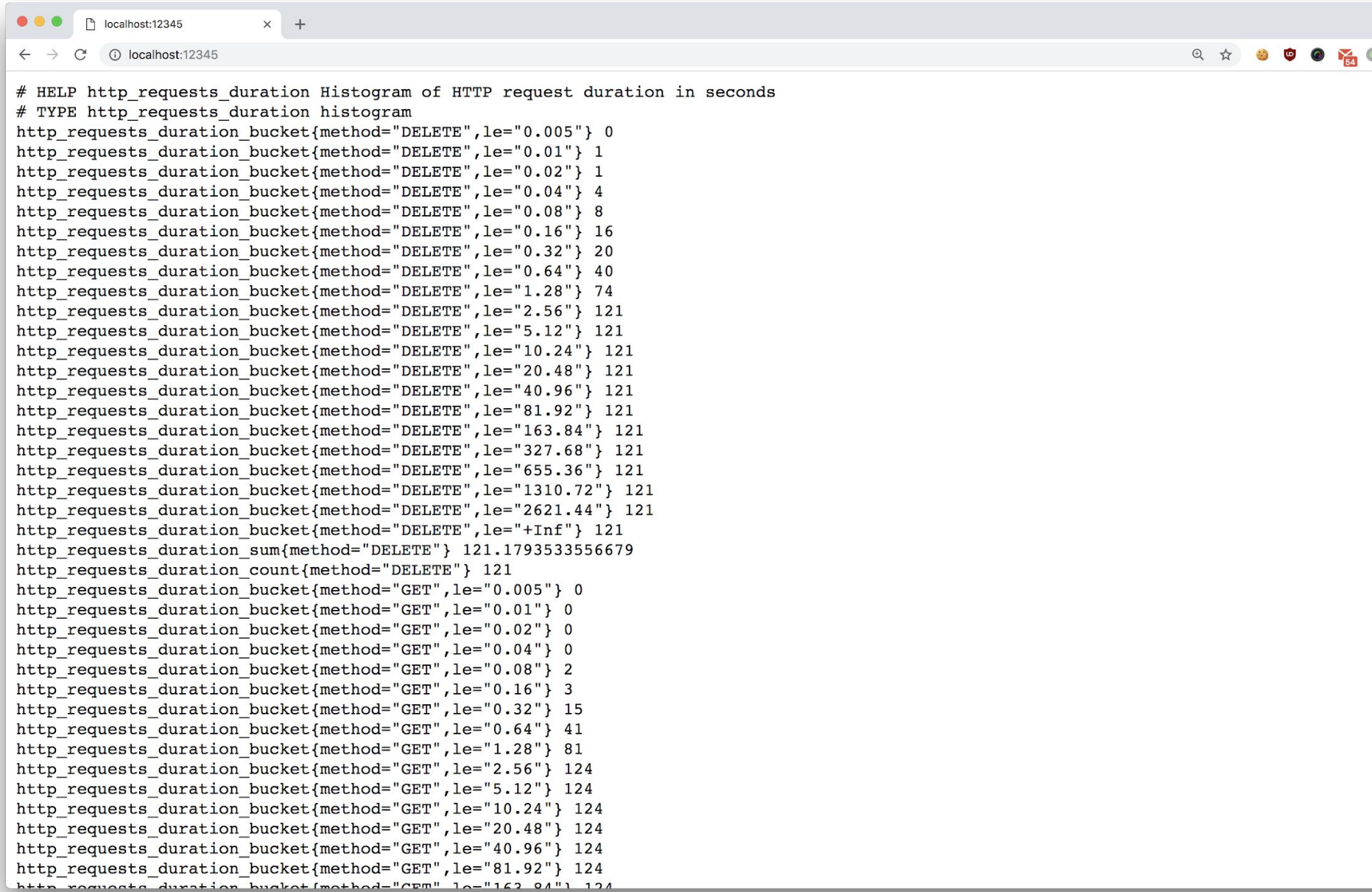
```rust
fn thread_simulate_requests() {
    let mut rng = rand::thread_rng();
    loop {
        // Simulate duration 0s ~ 2s
        let duration = rng.gen_range(0f64, 2f64);
        // Simulate HTTP method
        let method = ["GET", "POST", "PUT", "DELETE"].choose(&mut rng).unwrap();
        // Record metrics
        println!("{}\t{:.3}s", method, duration);
        REQUEST_DURATION.with_label_values(&[method]).observe(duration);
        // One request per second
        std::thread::sleep(std::time::Duration::from_secs(1));
    }
}
```

3. Push / Pull metrics for Prometheus

Example: metric service for pulling using hyper.

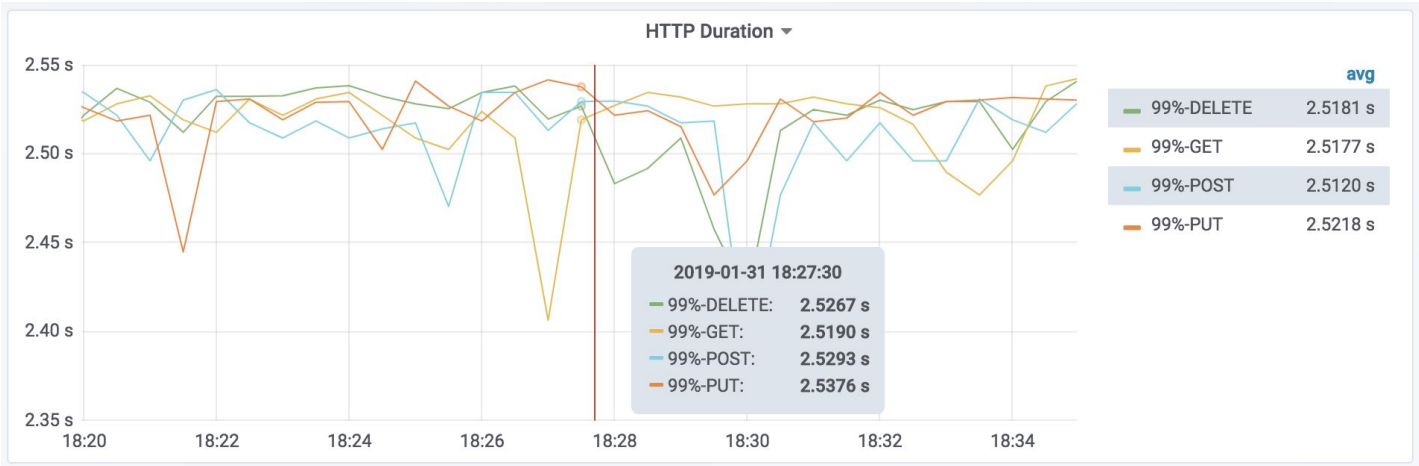
https://gist.github.com/breeswish/bb10bccd13a7fe332ef534ff0306ceb5

```
fn metric_service(_req: Request<Body>) -> Response<Body> {
    let encoder = TextEncoder::new();
    let mut buffer = vec![];
    let mf = prometheus::gather();
    encoder.encode(&mf, &mut buffer).unwrap();
    Response::builder()
        .header(hyper::header::CONTENT_TYPE, encoder.format_type())
        .body(Body::from(buffer))
        .unwrap()
}
```


localhost:12345
localhost:12345
HELP http_requests_duration Histogram of HTTP request duration in seconds
TYPE http_requests_duration histogram
http_requests_duration_bucket{method="DELETE",le="0.005"} 0
http_requests_duration_bucket{method="DELETE",le="0.01"} 1
http_requests_duration_bucket{method="DELETE",le="0.02"} 1
http_requests_duration_bucket{method="DELETE",le="0.04"} 4
http_requests_duration_bucket{method="DELETE",le="0.08"} 8
http_requests_duration_bucket{method="DELETE",le="0.16"} 16
http_requests_duration_bucket{method="DELETE",le="0.32"} 20
http_requests_duration_bucket{method="DELETE",le="0.64"} 40
http_requests_duration_bucket{method="DELETE",le="1.28"} 74
http_requests_duration_bucket{method="DELETE",le="2.56"} 121
http_requests_duration_bucket{method="DELETE",le="5.12"} 121
http_requests_duration_bucket{method="DELETE",le="10.24"} 121
http_requests_duration_bucket{method="DELETE",le="20.48"} 121
http_requests_duration_bucket{method="DELETE",le="40.96"} 121
http_requests_duration_bucket{method="DELETE",le="81.92"} 121
http_requests_duration_bucket{method="DELETE",le="163.84"} 121
http_requests_duration_bucket{method="DELETE",le="327.68"} 121
http_requests_duration_bucket{method="DELETE",le="655.36"} 121
http_requests_duration_bucket{method="DELETE",le="1310.72"} 121
http_requests_duration_bucket{method="DELETE",le="2621.44"} 121
http_requests_duration_bucket{method="DELETE",le="+Inf"} 121
http_requests_duration_sum{method="DELETE"} 121.1793533556679
http_requests_duration_count{method="DELETE"} 121
http_requests_duration_bucket{method="GET",le="0.005"} 0
http_requests_duration_bucket{method="GET",le="0.01"} 0
http_requests_duration_bucket{method="GET",le="0.02"} 0
http_requests_duration_bucket{method="GET",le="0.04"} 0
http_requests_duration_bucket{method="GET",le="0.08"} 2
http_requests_duration_bucket{method="GET",le="0.16"} 3
http_requests_duration_bucket{method="GET",le="0.32"} 15
http_requests_duration_bucket{method="GET",le="0.64"} 41
http_requests_duration_bucket{method="GET",le="1.28"} 81
http_requests_duration_bucket{method="GET",le="2.56"} 124
http_requests_duration_bucket{method="GET",le="5.12"} 124
http_requests_duration_bucket{method="GET",le="10.24"} 124
http_requests_duration_bucket{method="GET",le="20.48"} 124
http_requests_duration_bucket{method="GET",le="40.96"} 124
http_requests_duration_bucket{method="GET",le="81.92"} 124

4. Visualization

```
histogram_quantile(
    0.99,
    sum(rate(http_requests_duration_bucket[1m]))
    by (le, method)
)
```



PingCAP

How Rust Shines: Safe

Why we care about safety

TiKV, our distributed KV database, need to report metrics.

Safety is critical:

- We don't want crashes
- We don't want data corruption

Case Study: Type-safe labels

Background

You can define and record labels for metric vectors:

```
http_requests
method=POST,ip=192.168.0.1,path=/api
```

Then you can query metrics for specific labels:

- How many requests come from `192.168.0.1`?
- How long 99% requests to `/api` take?

Case Study: Type-safe labels

```
http_requests
method=POST,ip=192.168.0.1,path=/api
```

Define

```rust
let counter = CounterVec::new(
    Opts::new("http_requests", "help"),
    &["method", "ip", "path"],
).unwrap();
```

Record

```rust
counter.with_label_values(
    &["POST", "192.168.0.1", "/api"]
).inc();
```

Case Study: Type-safe labels

```
http_requests
method=POST,ip=192.168.0.1,path=/api
```

Define

```
let counter = CounterVec::new(
    Opts::new("http_requests", "help"),
    &["method", "ip", "path"],
).unwrap();
```

Record

```
counter.with_label_values(
    &["POST", "192.168.0.1", "/api"]
).inc();
```

Restriction

Equal number of label pairs

Case Study: Type-safe labels

Challenge

Equal number of label pairs

Solution 1

Check length at runtime, panics / throws error if not equal

Cons:

- The code may be hidden in a branch, not covered in tests → error in production
- Runtime cost

Case Study: Type-safe labels

Solution 2

Utilize types to enforce label length

```
trait Label { ... }
impl Label for [String; 1] { ... }
impl Label for [String; 2] { ... }
impl Label for [String; 3] { ... }

fn new<T: Label>(labels: T) -> CounterVec<T> { ... }

impl<T: Label> CounterVec<T> {
   fn with_label_values(values: T) -> ... { ... }
}

// Usage
let counter: CounterVec<[String; 2]> = CounterVec::new(
    [foo, bar]
);
counter.with_label_values([foo_value, bar_value])...
// This will not compile:
counter.with_label_values([foo_value])...
```

Case Study: Type-safe labels

Improvement

We further want...

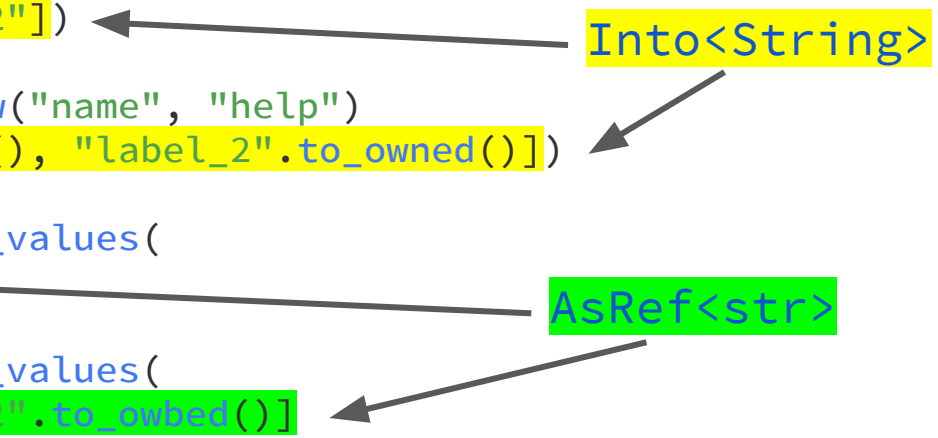
- `[Into<String>; N]` when defining,
- `[AsRef<str>; N]` when recording,
- `N` to be many values (e.g. `0..32`), but also DRY

Reference Implementation | Available in upcoming 1.0

https://github.com/pingcap/rust-prometheus/blob/ng/src/counter.rs

Case Study: Type-safe labels

```rust
// These will compile:
let counters = CounterBuilder::new("name", "help")
    .build_vec(["label_1", "label_2"])
    .unwrap();
let counters = CounterBuilder::new("name", "help")
    .build_vec(["label_1".to_owned(), "label_2".to_owned()])
    .unwrap();
let counter = counters.with_label_values(
    ["value_1", "value_2"]
);
let counter = counters.with_label_values(
    ["value_1".to_owned(), "value_2".to_owbed()]
);

// This will not compile:
let counter = counters.with_label_values(["1"]]);
// This will not compile as well:
let counter = counters.with_label_values(["1", "2", "3"]]);
```



Other cases

- `Send` and `Sync` markers
  - Consider thread local variables: `!Send`
  - We will see a `!Sync` example soon
- `#[must_use]`
  - Consider a timer records elapsed time when it is dropped

PingCAP

How Rust Shines: Safe

Why we care about performance

Metric recording is very frequent:

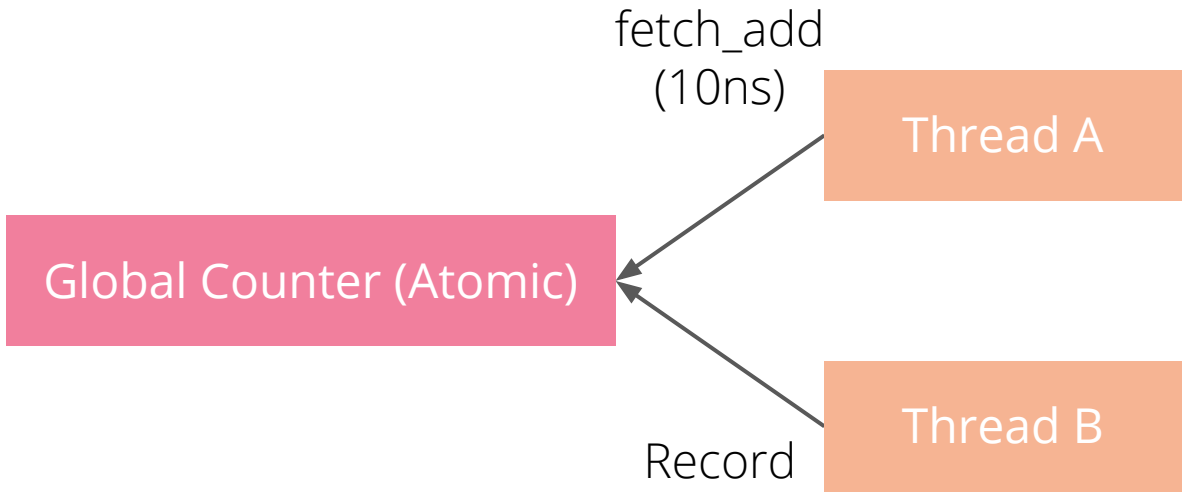
- We record a lot of metrics: duration, scanned keys, skipped keys, etc
- We record metrics for all operations: Get, Put, Scan, etc

The overhead of recording metrics should be small, so that we can know what is happening without sacrificing performance.

Case Study: Local `!Sync` metrics

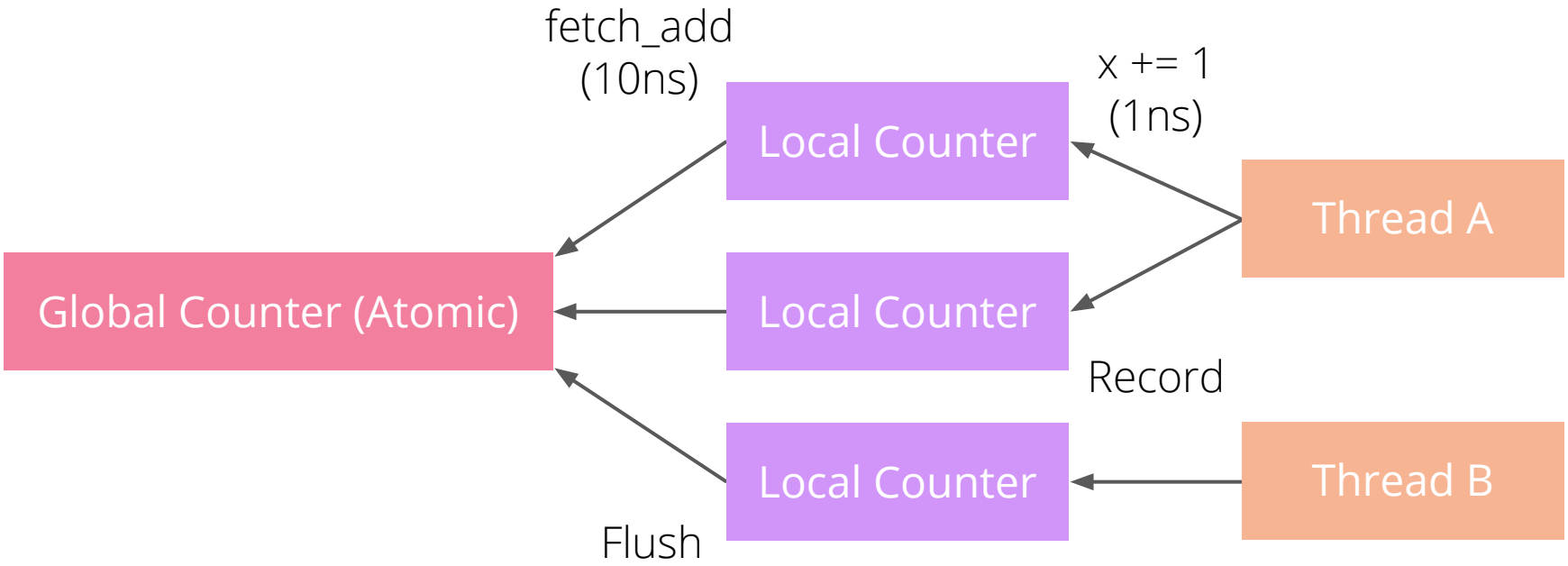
Global metrics are atomic variables, so that we can update it from everywhere, e.g. multiple threads.

Cons: Not so fast.



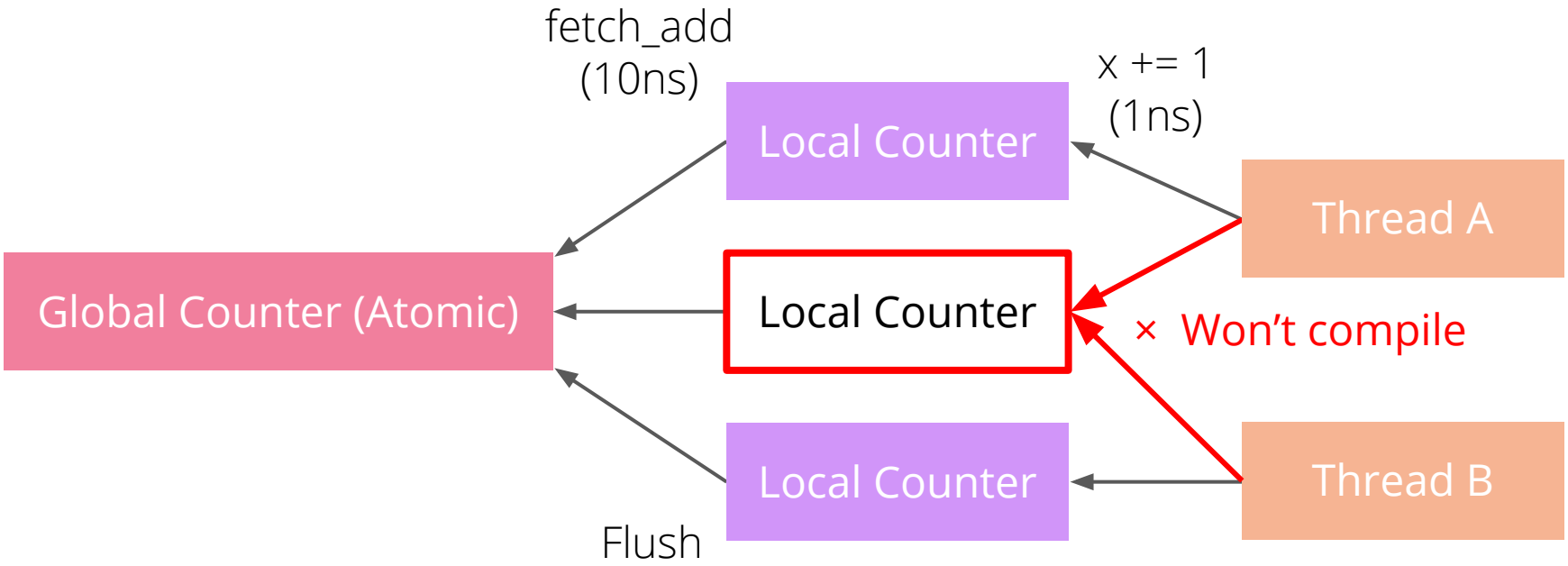
Case Study: Local `!Sync` metrics

Local metrics are `!Sync`. They are just local variables that flushing back to the corresponding global metric periodically.



Case Study: Local `!Sync` metrics

Pros: Both fast (*at local variable cost*) and safe (*!Sync*).



Case Study: Cache metric vectors

Background

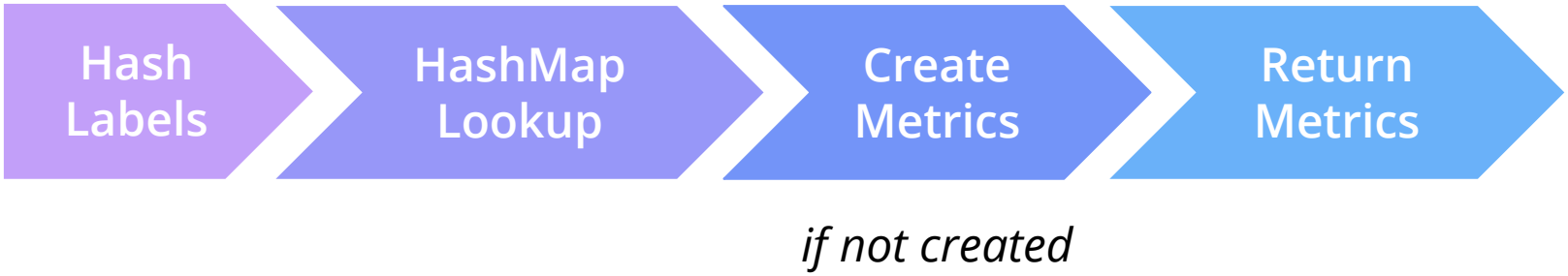
Metrics with different label values are counted independently:

```
let counters = CounterVec::new(
    Opts::new("name", "help"),
    &["method", "path"],
).unwrap();

counters.with_label_values(&["POST", "/"]).inc();
counters.with_label_values(&["GET", "/"]).inc();
counters.with_label_values(&["GET", "/api"]).inc();
counters.with_label_values(&["POST", "/"]).inc();

assert_eq!(counters.with_label_values(
    &["POST", "/"]).get(), 2.0
);
assert_eq!(counters.with_label_values(
    &["GET", "/api"]).get(), 1.0
);
```

Case Study: Cache metric vectors

Background

The internal of `with_label_values`:



Case Study: Cache metric vectors

Optimize

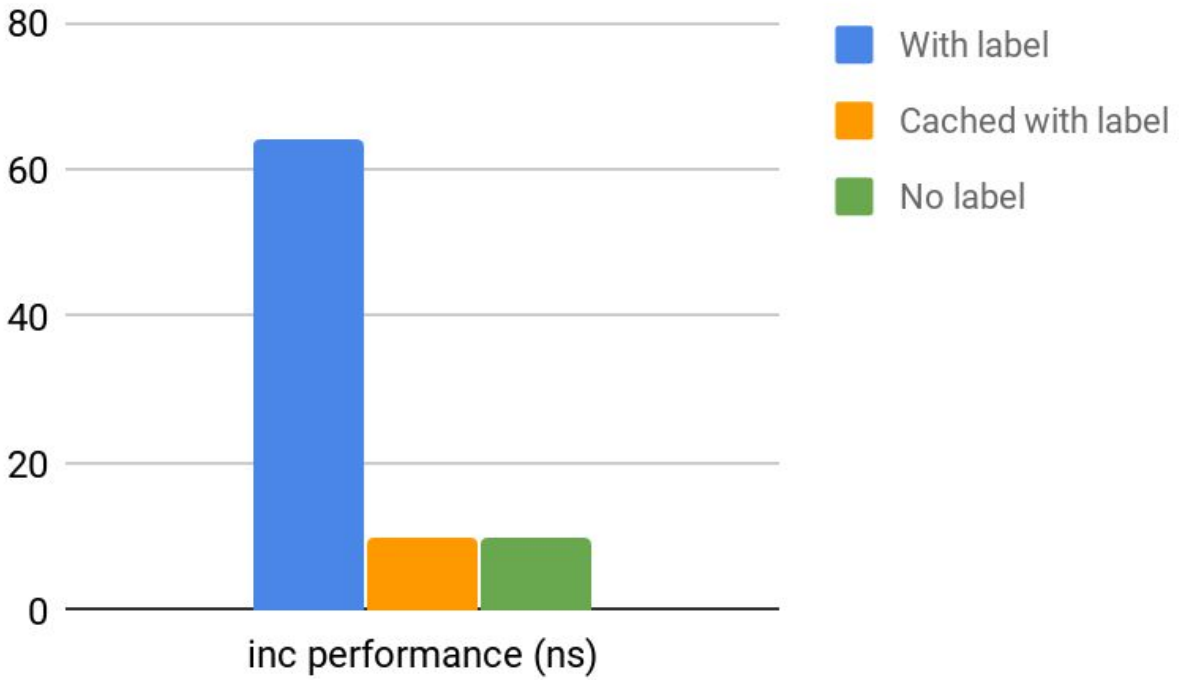
So if possible labels are known at compile time, instead of....

```
for i in 0..100 {
    // For each request:
    counters.with_label_values(&["GET", "/api"]).inc();
}
```

We can cache metric vectors:

```
let get_api_counter = counters.with_label_values(&["GET",
"/api"]);
for i in 0..100 {
    // For each request:
    get_api_counter.inc();
}
```

Case Study: Cache metric vectors



Case Study: Cache metric vectors

Challenge Not DRY.

Services
txn_get
txn_batch_get
txn_prewrite
txn_commit
txn_scan
txn_batch_scan
raw_get
raw_put
raw_delete
raw_scan

Code

```
let h_txn_get = counters.with_label_values(&["txn_get"]);
let h_txn_batch_get = counters.with_label_values(&["txn_batch_get"]);
let h_txn_prewrite = counters.with_label_values(&["txn_prewrite"]);
let h_txn_commit = counters.with_label_values(&["txn_commit"]);
let h_txn_scan = counters.with_label_values(&["txn_scan"]);
let h_txn_batch_scan = counters.with_label_values(&["txn_batch_scan"]);
let h_raw_get = counters.with_label_values(&["raw_get"]);
let h_raw_put = counters.with_label_values(&["raw_put"]);
let h_raw_delete = counters.with_label_values(&["raw_delete"]);
let h_raw_scan = counters.with_label_values(&["raw_scan"]);
```

Case Study: Cache metric vectors

Challenge Not DRY.

Services	Status
txn_get	success
txn_batch_get	fail
txn_prewrite	
txn_commit	
txn_scan	
txn_batch_scan	
raw_get	
raw_put	
raw_delete	
raw_scan	

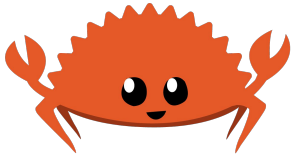
```
let h_txn_get_success = counters.with_label_values(&["txn_get", "success"]
let h_txn_get_fail = counters.with_label_values(&["txn_get", "fail"]);
let h_txn_batch_get_success = counters.with_label_values(&["txn_batch_get"
let h_txn_batch_get_fail = counters.with_label_values(&["txn_batch_get", "
let h_txn_prewrite_success = counters.with_label_values(&["txn_prewrite",
let h_txn_prewrite_fail = counters.with_label_values(&["txn_prewrite", "fa
let h_txn_commit_success = counters.with_label_values(&["txn_commit", "suc
let h_txn_commit_fail = counters.with_label_values(&["txn_commit", "fail"]
let h_txn_scan_success = counters.with_label_values(&["txn_scan", "success
let h_txn_scan_fail = counters.with_label_values(&["txn_scan", "fail"]);
let h_txn_batch_scan_success = counters.with_label_values(&["txn_batch_sca
let h_txn_batch_scan_fail = counters.with_label_values(&["txn_batch_scan",
let h_raw_get_success = counters.with_label_values(&["raw_get", "success"]
let h_raw_get_fail = counters.with_label_values(&["raw_get", "fail"]);
let h_raw_put_success = counters.with_label_values(&["raw_put", "success"]
let h_raw_put_fail = counters.with_label_values(&["raw_put", "fail"]);
let h_raw_delete_success = counters.with_label_values(&["raw_delete", "suc
let h_raw_delete_fail = counters.with_label_values(&["raw_delete", "fail"]
let h_raw_scan_success = counters.with_label_values(&["raw_scan", "success
let h_raw_scan_fail = counters.with_label_values(&["raw_scan", "fail"]);
```

PingCAP

Case Study: Cache metric vectors

Solution

Rust Macros



Services	Status
txn_get	success
txn_batch_get	fail
txn_prewrite	
txn_commit	
txn_scan	
txn_batch_scan	
raw_get	
raw_put	
raw_delete	
raw_scan	

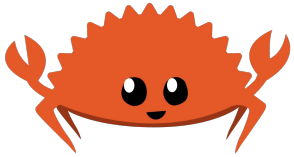
Code

```
make_static_metric! {
    pub struct MyStaticCounterVec: Counter {
        "services" => {
            txn_get, txn_batch_get, txn_prewrite,
            txn_commit, txn_scan, txn_batch_scan,
            raw_get, raw_put, raw_delete, raw_scan,
        },
        "status" => {
            success, fail,
        },
    }
}
```

Case Study: Cache metric vectors

Solution

Rust Macros



Services	Status	Usage
txn_get	success	
txn_batch_get	fail	
txn_prewrite		
txn_commit		
txn_scan		
txn_batch_scan		
raw_get		
raw_put		
raw_delete		
raw_scan		

```
let m = MyStaticCounterVec::from(&counter_vec);
m.txn_get.success.inc();
m.raw_put.fail.inc();
```

PingCAP

The Static Metric Macro Simplified

We want some macros that expand...

Invoke

```
make_metrics! {
  pub struct MyStaticMetric {
    foo, bar,
  }
}
```

Usage

```
fn main() {
  COUNTER
    .with_label_values(&["foo"])
    .inc();

   let m = MyStaticMetric::new();
   m.foo.inc();

   assert_eq!(m.foo.get(), 2.0);
   assert_eq!(m.bar.get(), 0.0);
}
```

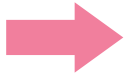
We want some macros that expand... into...

Invoke

```
make_metrics! {
  pub struct MyStaticMetric {
    foo, bar,
  }
}
```

Usage

```
fn main() {
  COUNTER
    .with_label_values(&["foo"])
    .inc();

   let m = MyStaticMetric::new();
   m.foo.inc();

   assert_eq!(m.foo.get(), 2.0);
   assert_eq!(m.bar.get(), 0.0);
}
```



Expanded

```
pub struct MyStaticMetric {
  foo: Counter,
  bar: Counter,
}

impl MyStaticMetric {
  pub fn new() -> Self {
    Self {
      foo: COUNTER.with_label_values(&["foo"]),
      bar: COUNTER.with_label_values(&["bar"]),
    }
  }
}
```

PingCAP

Or even some more complicated one in future?

Invoke

```
make_metrics! {
    pub struct MyStaticMetric {
        [ foo, bar ],
        [ user_a, user_b, user_c ],
        [ success, fail ],
    }
}
```

Expanded

```
pub struct MyStaticMetric {
  foo: MyStaticMetricInner2,
  bar: MyStaticMetricInner2,
}
pub struct MyStaticMetricInner2 {
  user_a: MyStaticMetricInner3,
  user_b: MyStaticMetricInner3,
  user_c: MyStaticMetricInner3,
}
pub struct MyStaticMetricInner3 {
  success: Counter,
  fail: Counter,
}
// ....
```

Declarative Macros won't work.

Let's use Procedural Macros!

PingCAP

Procedural Macros

Procedural macros allow creating syntax extensions as execution of a function.

- Function-like macros - `custom!(...)`
- Derive mode macros - `#[derive(CustomMode)]`
- Attribute macros - `#[CustomAttribute]`

Cargo.toml

```
[lib]
proc-macro = true
```

Provide `TokenStream` processing function

lib.rs

```rust
extern crate proc_macro;

use proc_macro::TokenStream;

#[proc_macro]
pub fn make_metrics(input: TokenStream) -> TokenStream {
    println!("{:?}", input);
    input
}
```

Let’s see what will happen!

examples/main.rs

```
make_metrics! {
  pub struct MyStaticMetric {
    foo, bar,
  }
}
```

Compiler Output

```
TokenStream [Ident { ident: "pub", span: #0
bytes(528..531) }, Ident { ident: "struct", span:
#0 bytes(532..538) }, Ident { ident:
"MyStaticMetric", span: #0 bytes(539..553) },
Group { delimiter: Brace, stream: TokenStream
[Ident { ident: "foo", span: #0 bytes(564..567) },
Punct { ch: ',', spacing: Alone, span: #0
bytes(567..568) }, Ident { ident: "bar", span: #0
bytes(569..572) }, Punct { ch: ',', spacing:
Alone, span: #0 bytes(572..573) }], span: #0
bytes(554..579) }]

error: expected `:`, found `,`
  --> examples/main.rs:30:12
   |
30 |         foo, bar,
   |            ^ expected `:`
```

Parse tokens using crate syn

examples/main.rs

```
make_metrics! {
  pub struct MyStaticMetric {
    foo, bar,
  }
}
```

lib.rs

```
#[derive(Debug)]
struct MetricDefinition {
  vis: Visibility,
  name: Ident,
  values: Vec<Ident>,
}

impl Parse for MetricDefinition {
  fn parse(input: ParseStream) -> Result<Self> {
    let vis = input.parse::<Visibility>()?;
    input.parse::<Token![struct]>()?;
    let name = input.parse::<Ident>()?;
    let content; braced!(content in input);
    let values = content
      .parse_terminated::<_, Token![,]>(Ident::parse)?
      .into_iter()
      .collect();
    Ok(Self { vis, name, values })
  }
}
```

Let’s see what we got!

examples/main.rs

```
make_metrics! {
  pub struct MyStaticMetric {
    foo, bar,
  }
}
```

lib.rs

```
#[proc_macro]
pub fn make_metrics(
  input: TokenStream
) -> TokenStream {
  let data = parse_macro_input!(
    input as MetricDefinition
  );
  println!("{:?}", data);
  unimplemented!()
}
```

Compiler Output

```
MetricDefinition {
    vis: Public(VisPublic { pub_token: Pub }),
    name: Ident { ident: "MyStaticMetric", span: #0
bytes(539..553) },
    values: [
        Ident { ident: "foo", span: #0 bytes(564..567) },
        Ident { ident: "bar", span: #0 bytes(569..572) }
    ]
}
```

PingCAP

Reassemble the code using crate `quote`

What we want

```
pub struct MyStaticMetric {
  foo: Counter,
  bar: Counter,
}

impl MyStaticMetric {
  pub fn new() -> Self {
    Self {
      foo: COUNTER.with_label_values(
        &["foo"]
      ),
      bar: COUNTER.with_label_values(
        &["bar"]
      ),
    }
  }
}
```

lib.rs

```
#[proc_macro]
pub fn make_metrics(
  input: TokenStream
) -> TokenStream {
  let data = parse_macro_input!(
    input as MetricDefinition
  );
  let vis = &data.vis;
  let name = &data.name;
  let values = &data.values;
  let expanded = quote! {
    #vis struct #name {
      #(#values: Counter,)*
    }
  };
  println!("{}", expanded);
  TokenStream::from(expanded)
}
```

Compiler Output

```
pub struct MyStaticMetric { foo : Counter ,
bar : Counter , }
```

PingCAP

Looks good. Let's continue.

What we want

```
pub struct MyStaticMetric {
  foo: Counter,
  bar: Counter,
}

impl MyStaticMetric {
  pub fn new() -> Self {
    Self {
      foo: COUNTER.with_label_values(
        &["foo"]
      ),
      bar: COUNTER.with_label_values(
        &["bar"]
      ),
    }
  }
}
```

lib.rs

```
let expanded = quote! {
  #vis struct #name {
    #(#values: Counter,)*
  }
  impl #name {
    pub fn new() -> Self {
      Self {
        #(#values: COUNTER.with_label_values(
            &["#values"]
        ),)*
      }
    }
  }
};
```

Compiler Output

```
pub struct MyStaticMetric { foo : Counter , bar : Counter , }
impl MyStaticMetric {
pub fn new (  ) -> Self { Self {
foo : COUNTER . with_label_values ( & [ "#values" ] ) ,
bar : COUNTER . with_label_values ( & [ "#values" ] ) , } } }
```

PingCAP

Ident -> LitStr

You can modify identifiers to whatever you want, e.g. adding suffix / prefix

What we want

```
pub struct MyStaticMetric {
  foo: Counter,
  bar: Counter,
}

impl MyStaticMetric {
  pub fn new() -> Self {
    Self {
      foo: COUNTER.with_label_values(
        &["foo"]
      ),
      bar: COUNTER.with_label_values(
        &["bar"]
      ),
    }
  }
}
```

lib.rs

```
let values_str = data.values
  .iter()
  .map(|ident| LitStr::new(
    &ident.to_string(),
    ident.span(),
  ));
let expanded = quote! {
  #vis struct #name {
    #(#values: Counter,)*
  }
  impl #name {
    pub fn new() -> Self {
      Self {
        #(#values: COUNTER.with_label_values(
          &[#values_str]
        ),)*
      }
    }
  }
};
```

That's it!

What we want

```
pub struct MyStaticMetric {
  foo: Counter,
  bar: Counter,
}

impl MyStaticMetric {
  pub fn new() -> Self {
    Self {
      foo: COUNTER.with_label_values(
        &["foo"]
      ),
      bar: COUNTER.with_label_values(
        &["bar"]
      ),
    }
  }
}
```

Generated tokens

```
pub struct MyStaticMetric {
  foo : Counter ,
  bar : Counter ,
}

impl MyStaticMetric {
  pub fn new (  ) -> Self {
    Self {
      foo : COUNTER . with_label_values (
        & [ "foo" ]
      ) ,
      bar : COUNTER . with_label_values (
        & [ "bar" ]
      ) ,
    }
  }
}
```

Check out

Our toy macro:

https://gist.github.com/breeswish/a337f36b58b282d912a31cb5d3031a88

The macro implementation of rust-prometheus:

https://github.com/pingcap/rust-prometheus/tree/master/static-metric

PingCAP

Future Plans

1.0 release (soon!)

- 2018 Edition
- Cleaner API
- Type-safe labels
- Small library size
- (maybe) `#[no_std]` compatible

Future

- Support Prometheus Summary
- Core-local metrics to be more efficient
- Easier to use pulling handler
- More to come!

Contributions are appreciated!

Thank You !

Learn more:

pingcap/rust-prometheus
pingcap/tidb
tikv/tikv

We are hiring!

hire@pingcap.com

We are:
***Remote** friendly,*
*Open Source **First**,*
***R&D** heavy*
***Rust**/Go/C++*

PingCAP