

计算机前沿技术丛书

# Rust 实战项目开发

朱伟 著

机械工业出版社

# 附录

## 附录 A: Rust edition 版本演化

Rust 语言遵循每六周发布一次更新版本，这也就意味着开发者每隔一段时间就可以获得不同版本的新功能或新特性。相比其它编程语言，Rust 更新速度要快得多，但这也表明每次更新都会更小、更稳定，一些微小的功能特性经过不断完善，可能在新版本中加入进来。在 Rust 2015 未稳定之前，Rust 语言几乎每天都在变化，这使得开发人员使用 Rust 编写软件变得非常困难，学习和维护成本相对来说比较高。随着 Rust 1.0 (2012 年) 和 Rust 2015 edition (2015 年 5 月份) 发布之后，语言本身致力于向后兼容，为开发者构建和维护项目打下坚实的基础。也就是说，从 Rust 1.0 发行版开始，一旦某个功能特性在稳定版上发布，将在所有将来的发行版中都支持该功能。然而，有时候在语法层面会进行小的更改是必要的，这时 edition (版本) 概念就发挥了重要作用。

Rust 2015 edition 是 Rust 语言所提供的众多特性已逐步走向稳定的一个版本，可以投入生产环境。因此，从 Rust 1.0 之后，它是默认版本。换句话说，如果你在 Cargo.toml 文件中未指定 edition 时，默认版本是 2015。当然，当我们使用 Cargo 工具创建 Rust 应用程序或组件库时，Cargo.toml 文件的 edition 默认是 rustup 工具所安装的 Rust edition 版本号，可能是 2018，也可能是 2021，甚至更高的版本号。

随着 Rust 官方团队和社区不断对 Rust 语言打磨和完善，在 2018 年 12 月 6 日发布了 Rust 2018 edition。这一版本主要目的是提高开发效率，包括提供编译器性能、推出一系列新的语言特性，以及改进 Rust 工具链、类库和文档等方面。Rust 2018 edition 发布，标志着 Rust 生产力提升方面取得了显著的进步，包括 impl trait、macros 2.0 变化、main 函数和 tests 中使用 ? 简写模式进行错误处理、SIMD 更快的计算支持、panic! 终止程序、async/await 关键字 (Rust 1.39 版本引入的) 和模块系统的变化，以及更加智能化的借用检查机制等等。此外，Rust 2018 在一定程度上放松了其稳定性保证，以支持可能破坏之前 Rust 2015 代码的语言规则，例如引入了 try 关键字，这可能会和某些函数名字或变量产生冲突。当开发者选择 2018 edition 时，就代表了开发者愿意接收 Rust 的版本内部变化和功能特性。为了帮助开发者从 Rust 2015 过度到 Rust 2018，Rust 提供了选择加入的功能，通过在项目中的 Cargo.toml 文件中添加 edition = "2018" 来实现。应用程序在编译时，一些废弃的或需要更新的特性，编译器都会发出对应的

提示，以辅助开发者进行修改或升级。Rust 2015 和 Rust 2018 的更多细节，你可以参考 Rust 版本指南：<https://doc.rust-lang.org/edition-guide/index.html>。

我在写本书时，使用 `rustup` 工具安装的 Rust edition 版本号为 2021。因此，本书中所有 Rust 应用程序或组件库所对应的 Rust edition 版本号都是 2021。Rust 2021 edition 于 2021 年 10 月 21 日发布，这一版本随着 Rust 1.56.0 稳定版推出，标志着 Rust 社区在版本更新方面又取得了新的突破。该版本的 Rust 设计注重实用性，旨在为开发者提供更加便捷和高效的编程体验。此外，随着 Rust 2021 的发布，Rust 社区还计划了一序列重要的新特性，以进一步提升语言的稳定性和实用性。开发者可以通过 `rustup` 工具升级对应的 Rust 工具链，例如：执行 `rustup update stable` 命令时，它会将当前 Rust 版本升级到最新的 Rust 版本，让开发者享受更多 Rust 新版本所带来的特性和功能。

在 Rust 官方 2022 年 2 月份发布的 2021 年 Rust Survey 调查报告显示：近万名受访者中，有 83% 的人认为在生产环境中使用 Rust 语言十分具有挑战性，主要体现在 Rust 学习曲线太陡峭、门槛高。开发者不仅要理解 Rust 各种新概念、新特性，而且还需要把具体的实现精确到很多细枝末节之处，例如：所有权、生命周期标注、借用检查机制等，需要开发者理解它们是如何工作的，才能让写出的 Rust 代码通过编译。也正是这个缘由，Rust 官方团队和社区在 Rust 2024 edition 设计路线图中，将进一步简化程序，使得开发者只需要处理其领域的固有复杂性，而不再去处理 Rust 的其它复杂性。Rust 官方团队希望开发者在使用 Rust 的过程中应该“不仅仅是可能，还需要不复杂且使用起来非常愉悦的编程”，特别是嵌入式、系统编程和异步编程等领域。为了实现这一愿景，Rust 官方团队大概有四个具体目标，如下所示：

- 更精确的分析：通过改进借用检查器、类型推理等，使编译器能够更好地识别代码是否正确。识别并消除 "boilerplate" 模式，如到处复制粘贴同一组 `where`。
- 开发人员应该能够更轻松、更直接地表达代码的意图。一方面可以通过语法糖的形式（如 `let-else`），另一方面可能意味着扩展类型系统。
- 改进异步支持：将 `async-await` 支持扩展到目前的 "MVP" 之外，包括 `traits` 中的 `async fns`、`async drop` 等功能。
- 让 `dyn Trait` 更有用处。拓宽可用于 `dyn` 的特性集，使使用 `dyn` 的工作更接近于使用泛型的工作。

除了上述这些目标之外，Rust 官方语言团队希望建立一些功能特性，使得库的作者能够更好地服务于它们的用户，无论是通过帮助管理功能的生命周期，还是通过扩大库的功能。同时，他们还希望能够在 Rust 生态中进行更多的探索和尝试，并且能够将 Rust 代码从生态稳定的第三方库迁移到标准库中来，例如 LazyCell 和 LazyLock 的引入，旨在提供数据的延迟初始化功能。另外，他们也希望增强 Rust 的互操作性，例如 FFI 调用，让 Rust 开发者能够轻松地使用 Rust 语言和其它语言互操作。最后，他们也希望开发者能够很容易地识别出团队正在积极开展那些工作，以及这些工作取得的进展。他们希望每一个跟踪问题都能清楚地识别出需要哪些步骤来推动一些特定功能的完成，并确保这些步骤对潜在的贡献值来说是足够清楚的。

从 Rust edition 版本演化的历程可以看出，Rust 官方团队非常希望 Rust 这一门语言具有更低的学习曲线、给予 Rust 库更好地连接生态，以及进一步发展和壮大 Rust 语言，让 Rust 能够为更多的开发者赋能，写出更加安全、高效率、高性能的软件。

## 附录 B: Rust tokio 运行时调度机制

在 Rust 异步编程领域，tokio 作为一个备受瞩目的运行时库，不仅提供了强大的功能，而且具有优秀的性能和可拓展性。tokio 的 Runtime（运行时）是实现异步操作的关键部分。tokio Runtime 和传统的同步编程模型不同，它提供了一种新的调度机制来处理并发和异步任务。Runtime 负责管理异步操作的生命周期，它提供了驱动器、调度器和计时器三个组件，使得开发者能够轻松创建和执行异步任务。这三个组件的作用如下：

- 驱动器 (Driver)：它是一个处理 I/O 事件和运行任务的线程，负责管理 I/O 资源，并分发 I/O 事件给依赖这些资源的任务。在异步编程中，I/O 操作通常是阻塞的，有时候会阻塞程序运行。通过驱动的管理，开发者可以轻松处理 I/O 事件，而不影响其它任务执行。
- 调度器 (Scheduler)：它是一组管理执行 Future 的工作队列，负责任务调度，使得任务能够并发执行。它将任务丢入队列中，并在适当的时机执行它们。通过合理的调度，能够充分利用系统资源，提升了程序的执行效率。
- 计时器 (Timer)：它是 tokio Runtime 的一个辅件，提供了定时器功能，可以安排任务在指定的时间内执行。也就是说，计时器提供了一种方便的方式处理定时任务，例如：定时触发事件、定期实行任务等，它能够帮助开发者更好地控制任务的执行顺序和时间。

以下是一个简单的例子，展示如何使用 tokio 运行时。

```
use tokio::runtime::Runtime;

fn main() {
    // 创建 tokio runtime 运行时实例

    let rt = Runtime::new().expect("failed to new tokio runtime");

    // 执行 future，阻塞当前线程直到完成

    rt.block_on(async {
        println!("hello,world!");
    });
}
```

```

// 在 tokio 运行时中生成一个 future

let handler = rt.spawn(async {

    println!("rt spawn task");

});

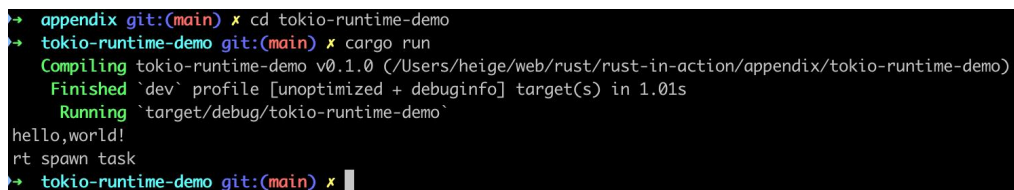
// 等待异步任务执行完毕

rt.block_on(handler).expect("failed to exec async task");

}

```

在上述示例代码中，我们首先创建了一个新的 tokio 运行时实例。然后，使用 `block_on` 方法在运行时内执行了一个异步任务，这个 `block_on` 方法会阻塞当前线程直到给定的异步任务完成。接着，使用 `spawn` 方法在 tokio 运行时中生成一个 future，该方法返回值是一个 `JoinHandle<T>`，它位于 `tokio::runtime::task::join` 模块中。这个 `JoinHandle<T>` 和标准库中 `std::thread::spawn` 返回值几乎相同。当 `spawn` 方法被调用时，提供的 future 将立即在后台运行，即使你没有等待返回的 `JoinHandle`。最后，通过 `block_on` 方法等待异步任务执行完毕。上述示例代码运行效果，如下图 B1-1 所示。



```

→ appendix git:(main) ✗ cd tokio-runtime-demo
→ tokio-runtime-demo git:(main) ✗ cargo run
Compiling tokio-runtime-demo v0.1.0 (/Users/heige/web/rust/rust-in-action/appendix/tokio-runtime-demo)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.01s
Running `target/debug/tokio-runtime-demo`
hello, world!
rt spawn task
→ tokio-runtime-demo git:(main) ✗

```

图 B1-1 tokio 运行时运行效果

接下来，让我们再看一个简单的 tokio spawn 示例：

```

#[tokio::main]

async fn spawn(){

    // 使用 tokio::spawn 函数在 tokio 运行时中生成一个 future,

    // 然后使用 await 关键字等待异步任务执行完毕。

    tokio::spawn(async{

        println!("hello, world!");

    }).await;

}

```

在上述代码中，使用#[tokio::main]属性宏将异步任务标记为 tokio 运行时执行。这个属性宏可以帮助用户设置异步运行时，而不需要开发者直接使用 Runtime 或 Builder 方式执行异步任务。上述代码实际上等价于下面不使用#[tokio::main]属性宏的代码：

```
fn main(){  
    // 通过 tokio runtime 创建了一个 tokio 运行时实例  
    tokio::runtime::Builder::new_multi_thread()  
        .enable_all()  
        .build()  
        .unwrap()  
        .block_on(async {  
            println!("hello,world!");  
        });  
}
```

在上述等价的代码片段中，首先使用 tokio::runtime::Builder::new\_multi\_thread 方法创建了一个多线程调度的 builder 对象，然后调用 build 方法（Builder 设计模式）创建一个 tokio 运行时实例。然后，通过 block\_on 方法等待异步任务执行完毕。从这个等价的代码片段可以看出，当我们使用 tokio::spawn 函数启动一个新的异步任务时，tokio 会在其内部的线程池中为这个任务分配一个执行线程，而不是在当前线程中直接执行。这种模型使得异步任务以非阻塞的方式运行，提高了系统资源利用率和响应速度。此外，tokio 的 spawn 函数并不保证异步任务执行完成，当运行时结束时，所有未完成的任务都会被丢弃。因此，需要使用 await 关键字驱动 Future 向前推进直到完成。在使用 tokio 时需要注意这一点，以确保程序的正确性和稳定性。

在我们熟悉了 tokio 基本用法后，接下来，让我们继续了解一下 tokio 运行时调度机制，如下图 B1-2 所示：

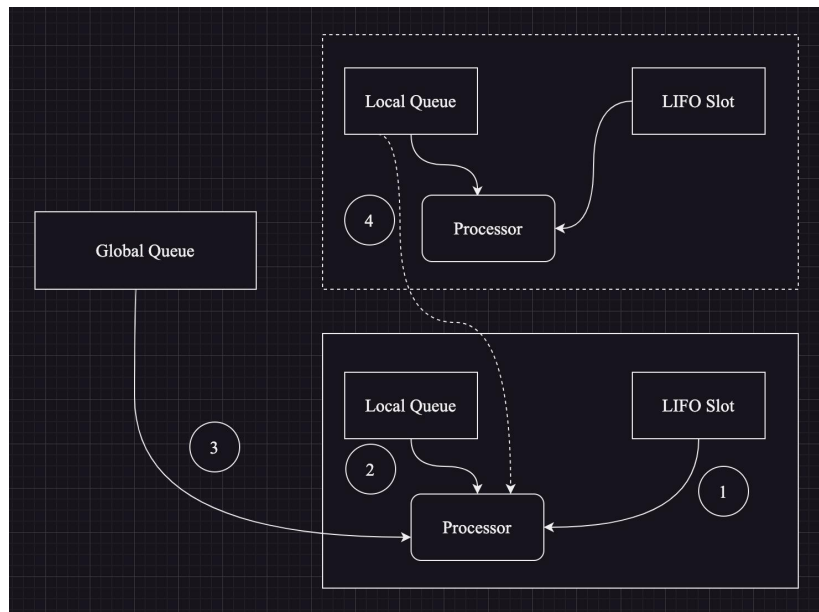


图 B1-2 tokio 运行时调度机制

在图 B1-2 中的 Local Queue (本地队列)、LIFO Slot (后进先出任务槽)、Processor (处理器) 都用于存储待处理的 Task (异步执行任务)。tokio Runtime 可以包含多个 Processor 和 LIFO Slot (目前 tokio 设计的 LIFO Slot 只能存放一个任务待执行), 所有的 Processor 共享 Global Queue。其中每个 Processor 都有自己的 Local Queue 本地队列存放任务。假设这个 Local Queue 的大小是 256 的话。如果在调度过程中超过了 256 的话, tokio 就会把 Processor 上面一半的 Tasks 任务移动到 Global Queue 全局队列上面。此时, 每个 Processor 都可以从全局队列中获取任务执行。当 Processor 运行完当前 Task 之后, 它会尝试以下顺序获取新的 Task 并继续运行:

- 1) 先从 LIFO Slot 中获取异步任务执行, 如果没有获取到任务, 就进入下一步;
  - 2) 从 Local Queue 中获取任务执行, 如果本地队列为空, 就进入下一步;
  - 3) 从 Global Queue 中获取一部分 Task 来执行。如果没有任务可执行, 就进入下一步;
  - 4) 从其它 Processor 处理器的 Local Queue 本地队列中获取一些任务, 继续执行。
- 如果当前 Processor 获取不到 Task 执行, 它对应的线程就会休眠, 等待下一次被唤醒。

tokio 这种任务调度机制, 采用 Local Queue 和 LIFO Slot 的目的是为了避免过多的线程之间的 data race (数据竞争) 和 deadlock (死锁), 保证一定的 CPU 资源本地性调度。上图 B1-2 中的 Global Queue 目的是为了存放更多的异步任务, 当然它是具



有锁的，因此不可避免会带来一些性能损耗。更多 tokio 运行时的技术细节，你可以参考如下地址：

- 官方文档: <https://tokio.rs/#tk-lib-runtime>
- tokio 源码: <https://github.com/tokio-rs/tokio/tree/master/tokio>

总之，tokio 运行时是异步任务执行的核心，为 Rust 异步编程提供了强大的支持，简化了异步编程的复杂度，能够高效地管理和调度大量的异步任务，实现并发执行。

## 附录 C：Docker 中 Kafka 和 Pulsar 基本操作

在 8.2~8.3 章节中，我们已经通过 Docker 容器安装了 Kafka 和 Pulsar 服务，接下来，我将详细介绍在 Docker 环境下如何操作 Kafka 和 Pulsar 消息队列。

### C1.1 Kafka 基本操作

首先，我们执行如下命令启动 Kafka 服务：

```
cd part8/kafka/dockerconfig
```

```
sh bin/docker-run.sh
```

然后，执行 `docker exec -it my-kafka /bin/bash` 命令进入 Kafka 服务终端窗口中。随后，切换到 Kafka 安装目录，运行效果如下图 C1-1 所示：

```
+ dockerconfig git:(main) ✖ sh bin/docker-run.sh
"docker rm" requires at least 1 argument.
See 'docker rm --help'.

Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers
[+] Running 2/2
 ✓ Container my-kafka-zk Started 0.0s
 ✓ Container my-kafka Started 0.1s
+ dockerconfig git:(main) ✖ docker ps | grep kafka
e052b0ae80cc wurstmeister/kafka "start-kafka.sh" 19 seconds ago Up 18 seconds 0.0.0.0:9092->9092/tcp
bb72a093089c wurstmeister/zookeeper "/bin/sh -c '/usr/sb..." 19 seconds ago Up 18 seconds 22/tcp, 2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp my-kafka-zk
+ dockerconfig git:(main) ✖ docker exec -it my-kafka /bin/bash
root@e052b0ae80cc:/# cd /opt/kafka/
root@e052b0ae80cc:/opt/kafka# ls
LICENSE NOTICE bin config libs licenses logs site-docs
root@e052b0ae80cc:/opt/kafka# ls bin/
connect-distributed.sh kafka-dump-log.sh kafka-storage.sh
connect-mirror-maker.sh kafka-features.sh kafka-streams-application-reset.sh
connect-standalone.sh kafka-leader-election.sh kafka-topics.sh
kafka-acls.sh kafka-log-dirs.sh kafka-verifiable-consumer.sh
kafka-broker-api-versions.sh kafka-metadata-shell.sh kafka-verifiable-producer.sh
kafka-cluster.sh kafka-mirror-maker.sh trogdor.sh
kafka-configs.sh kafka-preferred-replica-election.sh windows
kafka-console-consumer.sh kafka-producer-perf-test.sh zookeeper-security-migration.sh
kafka-console-producer.sh kafka-reassign-partitions.sh zookeeper-server-start.sh
kafka-consumer-groups.sh kafka-replica-verification.sh zookeeper-server-stop.sh
kafka-consumer-perf-test.sh kafka-run-class.sh zookeeper-shell.sh
kafka-delegation-tokens.sh kafka-server-start.sh
kafka-delete-records.sh kafka-server-stop.sh
root@e052b0ae80cc:/opt/kafka#
```

图 C1-1 docker 环境下的 kafka 服务

从图 C1-1 中看出，Kafka 安装目录中有 bin、config、libs、logs 等目录。其中在 bin 目录下面有各种 kafka 命令操作的 shell 脚本，通过这些脚本可以辅助我们更好地操作和管理 kafka。下面是 Kafka 常用的命令操作：

#### # 创建 topic

```
./bin/kafka-topics.sh --create --topic test --bootstrap-server localhost:9092
```

当终端输出“Created topic test”就表示 topic 创建成功。

#### # 发送消息

```
./bin/kafka-console-producer.sh --topic test --bootstrap-server localhost:9092
```

在交互窗口中输入消息并回车就可以发送消息，效果如图 C1-2 所示：

```
root@e052b0ae80cc:/opt/kafka# ./bin/kafka-console-producer.sh --topic test --bootstrap-server localhost:9092
>hello
>hello,kafka
>hello,rust
>hello,123
>
```

图 C1-2 kafka 发送消息到 topic

## # 消费消息

```
./bin/kafka-console-consumer.sh --topic test --from-beginning --bootstrap-server
```

localhost:9092

该命令需要在另一个终端中运行，效果下图 C1-3 所示：

```
root@e052b0ae80cc:/opt/kafka# ./bin/kafka-console-producer.sh --topic test --bootstrap-server localhost:9092
>hello
>hello,kafka
>hello,rust
>hello,123
>[]

docker (com.docker.cli)
Last login: Thu Apr  4 15:33:18 on ttys009
You have new mail.
➔ ~ docker exec -it my-kafka /bin/bash
root@e052b0ae80cc:/# cd /opt/kafka/
root@e052b0ae80cc:/opt/kafka# ./bin/kafka-console-consumer.sh --topic test --from-beginning --bootstrap-server localhost:9092
hello
hello,kafka
hello,rust
hello,123
```

图 C1-3 kafka 消费消息

从图 C1-3 中看出，我们在图 C1-2 终端中发送的消息已正常消费。如果你不想从消息开始位置消费消息，你可以去掉--from-beginning 参数，运行效果如图 C1-4 所示：

```
>hello,rust
>hello,123
>hello
>hello,kafka
>[]

docker (com.docker.cli)
root@e052b0ae80cc:/opt/kafka# ./bin/kafka-console-consumer.sh --topic test --bootstrap-server localhost:9092
hello,rust
hello,123
hello
hello,kafka
```

图 C1-4 kafka 实时监听消费消息

## # 查看和删除 topic 操作

```
./bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

```
./bin/kafka-topics.sh --delete --bootstrap-server localhost:9092 --topic test
```

执行这两个命令后，就可以看到 kafka 对应的 topic 列表以及删除后的效果，如下图 C1-5 所示：

```

root@e052b0ae80cc:/opt/kafka# ./bin/kafka-topics.sh --list --bootstrap-server localhost:9092
__consumer_offsets
test
root@e052b0ae80cc:/opt/kafka# ./bin/kafka-topics.sh --delete --bootstrap-server localhost:9092 --topic test
root@e052b0ae80cc:/opt/kafka# ./bin/kafka-topics.sh --list --bootstrap-server localhost:9092
__consumer_offsets
root@e052b0ae80cc:/opt/kafka#

```

图 C1-5 查看和删除 topic

如果你使用的 docker 本地 kafka 集群模式，你可以通过 `docker exec -it kafka1 /bin/bash` 进入容器中，通过 kafka 提供的 shell 脚本就可以运行相关命令，效果如图 C1-6 所示：

```

➔ dockerconfig git:(main) ✗ sh bin/docker-cluster-run.sh
d9d1a0b98714
64ee6f55e115
d9bdd1e709bd
6935c7563ff5
[+] Running 4/4
✔ Container my-kafka-zk Started 0.0s
✔ Container kafka2 Started 0.1s
✔ Container kafka1 Started 0.1s
✔ Container kafka3 Started 0.1s
➔ dockerconfig git:(main) ✗ docker exec -it kafka1 /bin/bash
root@ecfd586f8ae4:/# cd /opt/kafka
root@ecfd586f8ae4:/opt/kafka# ./bin/kafka-topics.sh --list --bootstrap-server localhost:9093

root@ecfd586f8ae4:/opt/kafka# ./bin/kafka-topics.sh --create --topic test --bootstrap-server localhost:9093
Created topic test.
root@ecfd586f8ae4:/opt/kafka# exit
exit
➔ dockerconfig git:(main) ✗ docker exec -it kafka2 /bin/bash
root@cd635e423b41:/# cd /opt/kafka
root@cd635e423b41:/opt/kafka# ./bin/kafka-topics.sh --list --bootstrap-server localhost:9094
test
root@cd635e423b41:/opt/kafka# ./bin/kafka-console-producer.sh --topic test --bootstrap-server localhost:9094
>hello,kafka cluster
>hello,rust
>hello,world
>^Croot@cd635e423b41:/opt/kafka#
root@cd635e423b41:/opt/kafka# ./bin/kafka-console-consumer.sh --topic test --from-beginning --bootstrap-server localhost:9094
hello,kafka cluster
hello,rust
hello,world

```

图 C1-6 kafka 集群模式相关命令操作

如果你想在 docker 环境下使用 Kafka 集群模式运行 Kafka 服务。首先，在 `dockerconfig` 目录下新建一个 `kafka-cluster` 目录，并在该目录下面新建 `docker-compose.yaml` 文件，详细的 docker 配置内容见如下链接。

<https://github.com/daheige/rust-in-action/blob/main/part8/kafka/dockerconfig/kafka-cluster/docker-compose.yaml>

接下来，在 `dockerconfig/bin` 下面新建一个 `docker-cluster-run.sh` 脚本，并添加如下内容：

```
#!/usr/bin/env bash
```

```
root_dir=$(cd "$(dirname "$0")"; cd ..; pwd)
```

```
# 删除本机原有的 kafka 容器
```

```
docker rm -f `docker ps -a | grep kafka | awk '{print $1}'`
```

```
cd $root_dir/kafka-cluster
```

`docker-compose up -d`

运行 `sh ./bin/docker-cluster-run.sh` 命令启动 kafka 集群服务，运行效果如图 C1-7 所示：

```
→ dockerconfig git:(main) ✖ sh bin/docker-cluster-run.sh
"docker rm" requires at least 1 argument.
See 'docker rm --help'.

Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers
[+] Running 5/5
✔ Network kafka-cluster_default      Created                                0.1s
✔ Container kafka-cluster-zookeeper-1 Started                                0.1s
✔ Container kafka1                    Started                                0.1s
✔ Container kafka3                    Started                                0.1s
✔ Container kafka2                    Started                                0.1s
→ dockerconfig git:(main) ✖ docker ps | grep kafka
88091f2f9dee   wurstmeister/kafka      "start-kafka.sh"          About a minute ago   Up About a minute   0.0.0.0:9095->9095/tcp
17787884e02c   wurstmeister/kafka      "start-kafka.sh"          About a minute ago   Up About a minute   0.0.0.0:9094->9094/tcp
b93c6adcb471   wurstmeister/kafka      "start-kafka.sh"          About a minute ago   Up About a minute   0.0.0.0:9093->9093/tcp
96ef67ed2acf   wurstmeister/zookeeper  "/bin/sh -c '/usr/sb..." About a minute ago   Up About a minute   22/tcp, 2888/tcp, 3888/
tcp, 0.0.0.0:2181->2181/tcp   kafka-cluster-zookeeper-1
→ dockerconfig git:(main) ✖
```

图 C1-7 kafka docker 本地集群模式

以上内容是 Kafka 的一些基本操作，kafka 具体的使用方法和细节还需要根据实际的需求和环境来进行调整。如果你想继续深入 kafka 使用方法以及底层架构设计细节，请参考官方文档：<https://kafka.apache.org>。

## C1.2 Pulsar 基本操作

以下内容是 Pulsar 消息队列基本操作：

- **创建 topic:** `bin/pulsar-admin topics create persistent://public/default/my-topic`
- **查看 topic list:** `bin/pulsar-admin topics list public/default`
- **发送消息:** `bin/pulsar-client produce my-topic --messages 'Hello rust!'`
- **消费消息:** `bin/pulsar-client consume my-topic -s 'my-subscription' -p Earliest -n 0`
- **删除 topic:** `bin/pulsar-admin topics delete persistent://public/default/my-topic`
- **查看 topic 状态:** `bin/pulsar-admin topics stats persistent://public/default/my-topic`

在 Pulsar docker 容器中执行上述命令。当然，这些命令也同样适用于集群版的 pulsar。更多 pulsar 命令操作，你可以直接参考 Pulsar CLI Tools Docs 官方在线文档：<https://pulsar.apache.org/reference/next/cli>。

## 附录 D: MacOS 系统安装 Qt 工具

在 MacOS 系统下，安装 Qt 工具和相关依赖步骤如下：

### 1) 安装 Qt6 必要的依赖

```
brew install llvm cmake make gcc mold clang-format
```

当看到如下图 D1-1 中的提示信息，就表明 Qt6 相关依赖安装成功。

```
➜ Installing mold
➜ Pouring mold-2.31.0.sonoma.bottle.tar.gz
➜ Caveats
Support for Mach-O targets has been removed.
See https://github.com/bluewhalesystems/sold for macOS/iOS support.
➜ Summary
📦 /usr/local/Cellar/mold/2.31.0: 405 files, 25MB
➜ Running 'brew cleanup mold'...
➜ Pouring clang-format-18.1.5.sonoma.bottle.tar.gz
📦 /usr/local/Cellar/clang-format/18.1.5: 10 files, 3MB
➜ Running 'brew cleanup clang-format'...
➜ Caveats
➜ cmake
To install the CMake documentation, run:
  brew install cmake-docs

Emacs Lisp files have been installed to:
  /usr/local/share/emacs/site-lisp/cmake
➜ make
GNU "make" has been installed as "gmake".
If you need to use it as "make", you can add a "gnubin" directory
to your PATH from your bashrc like:

  PATH="/usr/local/opt/make/libexec/gnubin:$PATH"
➜ mold
Support for Mach-O targets has been removed.
See https://github.com/bluewhalesystems/sold for macOS/iOS support.
```

图 D1-1 通过 brew 安装 Qt6 相关依赖

### 2) 使用 brew 安装 Qt 工具

```
brew install qt6
```

#下面的 qt-creator 可选，如果你需要使用 C++语言开发 Qt 项目，就需要安装它

```
brew install qt-creator
```

当看到下图 D1-2 中的提示信息，就表明 Qt 工具安装正常。

```
➜ Installing qt dependency: webp
➜ Pouring webp-1.4.0.sonoma.bottle.tar.gz
📦 /usr/local/Cellar/webp/1.4.0: 63 files, 2.5MB
➜ Installing qt
➜ Pouring qt-6.7.0_1.sonoma.bottle.tar.gz
➜ Caveats
You can add Homebrew's Qt to QtCreator's "Qt Versions" in:
  Preferences > Qt Versions > Link with Qt...
pressing "Choose..." and selecting as the Qt installation path:
  /usr/local
➜ Summary
📦 /usr/local/Cellar/qt/6.7.0_1: 15,057 files, 675.2MB
➜ Running 'brew cleanup qt'...
```

图 D1-2 MacOS 下安装 qt 提示

需要注意一点：使用 brew 安装 Qt 的版本可能是高版本的，在开发实际的 Qt 项目时，我们可以根据实际情况选择对应的 Qt 版本。从图 D1-2 中看出，安装好的 Qt 工具放在/usr/local/Cellar/qt 目录中。当我们切换到/usr/local/Cellar/qt 目录中，执行 ls 命令查看对应的 Qt 版本时，效果如下所示：

```
% ls
```

6.7.0\_1

3) Qt link 绑定 (这一步需要做, 不然 cxx-qt 找不到 qt 相关的路径)

```
brew link qt
```

4) 设置 Qt 相关的环境变量

vim ~/.bash\_profile 添加如下内容:

```
# 分别对 LDFLAGS CPPFLAGS PKG_CONFIG_PATH 配置
```

```
export QT_HOME=/usr/local/Cellar/qt/6.7.0_1 # qt 安装目录
```

```
# 请通过下面的方式设置 LDFLAGS
```

```
export LDFLAGS="$LDFLAGS -L$QT_HOME/lib"
```

```
export LDFLAGS="$LDFLAGS -L/usr/local/opt/llvm/lib"
```

```
export LDFLAGS="$LDFLAGS -L/usr/local/opt/llvm/lib/c++
```

```
-Wl,-rpath,/usr/local/opt/llvm/lib/c++"
```

```
# 请通过下面的方式设置 CPPFLAGS
```

```
export CPPFLAGS="$CPPFLAGS -I$QT_HOME/include"
```

```
export CPPFLAGS="$CPPFLAGS -I/usr/local/opt/llvm/include"
```

```
# 对于 PKG_CONFIG_PATH 设置, 如果系统中没有设置过
```

```
export PKG_CONFIG_PATH="$QT_HOME/lib/pkgconfig"
```

```
# 如果你执行 echo $PKG_CONFIG_PATH 命令, 输出结果不为空, 就通过通过下面  
的方式设置
```

```
# export PKG_CONFIG_PATH="$PKG_CONFIG_PATH:$QT_HOME/lib/pkgconfig"
```

```
export PATH="$QT_HOME/bin:$PATH"
```

```
export PATH="/usr/local/opt/llvm/bin:$PATH"
```

执行:wq 保存退出, 再执行 source 命令生效:

```
source ~/.bash_profile
```

接着, 执行 `qmake --version` 命令验证 Qt 工具是否安装成功。如果运行效果如下图 D1-3 所示, 就表明 Qt 工具安装成功。

```
+ ~ cd /usr/local/Cellar/qt/6.7.0_1
+ 6.7.0_1 ls
Frameworks      INSTALL_RECEIPT.json README.md      bin      include      lib      libexec      share
+ 6.7.0_1 qmake --version
QMake version 3.1
Using Qt version 6.7.0 in /usr/local/lib
+ 6.7.0_1
```

图 D1-3 qmake 运行效果

#### 5) qt-creator 配置 (可选)

如果你需要使用 C++ 语言开发 Qt 应用程序, 那么就需要打开 qt-creator 软件配置 qt link 的路径为 Qt 工具安装的完整绝对路径, 在这里是 `/usr/local/Cellar/qt/6.7.0_1` 目录。



## 附录 E: QA 问答系统相关内容

在本附录中，主要包括 QA 问答系统的 `protoc` 工具安装、配置文件内容和数据结构定义、`grpcurl` 工具安装和基本使用等内容。

### E1.1 `protoc` 工具安装

为了能够在 11.3 中快速使用 `tonic` 构建 gRPC 微服务接口，你需要安装 `protoc` 工具。`protoc` 工具的安装因操作系统不同，其安装方式不一样，你可以根据实际情况选择其中的一种方式即可。下面的内容是如何在 Linux Centos、Ubuntu 以及 MacOS 系统上面安装 `protoc` 工具的步骤。

#### **Centos 安装 `protoc` 工具:**

##### 1) 下载 `protobuf` 包

```
cd /usr/local/src/
```

```
sudo wget https://github.com/protocolbuffers/protobuf/archive/v3.15.8.tar.gz
```

##### 2) 安装相关依赖

```
sudo yum install gcc-c++ cmake libtool
```

##### 3) 解压 `protobuf` 包，并使用 `make` 编译安装

```
sudo mv v3.15.8.tar.gz protobuf-3.15.8.tar.gz
```

```
sudo tar zxvf protobuf-3.15.8.tar.gz
```

```
sudo mkdir /usr/local/protobuf
```

```
cd protobuf-3.15.8
```

```
./autogen.sh
```

```
sudo ./configure --prefix=/usr/local/protobuf
```

```
sudo make && sudo make install
```

##### 4) 查看安装 `protoc` 版本

```
cd /usr/local/protobuf/bin
```

```
./protoc --version
```

执行上面命令后，会输出：libprotoc 3.15.8

##### 5) 建立 `protoc` 软链接

```
sudo ln -s /usr/local/protobuf/bin/protoc /usr/bin/protoc
```

```
sudo chmod +x /usr/bin/protoc
```

### Ubuntu 系统安装 protoc 工具:

#### 1) 安装相关依赖

```
sudo apt-get install gcc cmake make libtool
```

#### 2) 安装 protobuf 工具包

```
sudo apt-get install libprotobuf-dev protobuf-compiler
```

#### 3) 查看安装版本

```
protoc --version
```

### MacOS 系统安装 protoc 工具:

#### 1) 安装相关依赖

```
brew install automake
```

```
brew install libtool
```

#### 2) 安装 protobuf

```
brew install protobuf
```

#### 3) 查看 protoc 版本

```
protoc --version
```

## E1.2 QA 配置文件和数据结构定义

在 qa-project/crates/qa-svc 目录中新建一个 app.yaml 文件, 并添加如下 yaml 配置。

```
app_debug: true # 是否开启调试模式
```

```
app_port: 50051 # grpc service 运行端口
```

```
metrics_port: 2338 # prometheus metrics port
```

```
graceful_wait_time: 3 # 平滑退出等待时间, 单位 s
```

```
# MySQL 数据库配置
```

```
mysql_conf:
```

```
  dsn: "mysql://root:root123456@127.0.0.1/qa_sys" # dsn 连接句柄信息
```

```
  max_connections: 100 # 最大连接数
```

```
  min_connections: 10 # 最小连接数
```

```
  max_lifetime: 1800 # 连接池默认生命周期, 单位 s
```

```
  idle_timeout: 300 # 空闲连接生命周期超时, 单位 s
```

```
  connect_timeout: 10 # 连接超时时间, 单位 s
```

#Pulsar 消息队列配置

pulsar\_conf:

addr: pulsar://127.0.0.1:6650

token: "" # pulsar auth token

#Redis 缓存配置

redis\_conf:

dsn: "redis://:@127.0.0.1:6379/0" # redis dsn 信息, 用于连接 redis

max\_size: 300 # 最大连接个数, 默认为 300

min\_idle: 3 # 最小空闲数, 默认为 3

max\_lifetime: 1800 # 过期时间, 默认为 1800s

idle\_timeout: 300 # 连接池最大生存期, 默认为 300s

connection\_timeout: 10 # 连接超时时间, 默认为 10s

# aes 加解密配置

aes\_key: "YiBX0z9WnJjsS5aNXmi0AeT1yTPZZJYa"

aes\_iv: "3ZQEpwP9DbK4h1Z0"

从这个 app.yaml 配置文件中可以看出, 我们定义了项目相关配置, 例如 app\_debug、app\_port、mysql\_conf、pulsar\_conf、redis\_conf 以及 aes 加解密的 aes\_key、aes\_iv 等配置信息。

当我们配置好对应的 app.yaml 文件后, 就可以在 crates/qa-svc/src/config/app.rs 文件中定义配置文件所对应的数据类型, 其核心代码片段如下:

```
// 定义项目中使用的 mysql pool 和 pulsar client
```

```
#[derive(Clone)]
```

```
pub struct AppState {
```

```
    pub mysql_pool: sqlx::MySqlPool,
```

```
    pub pulsar_client: Pulsar<TokioExecutor>,
```

```
    pub redis_pool: Pool<redis::Client>,
```

```
}
```

```

// AppConfig 项目配置信息

#[derive(Debug, PartialEq, Serialize, Deserialize, Default)]

pub struct AppConfig {

    pub mysql_conf: mysql::MysqlConf,

    pub pulsar_conf: xpulsar::PulsarConf,

    pub redis_conf: xredis::RedisConf,

    pub app_port: u16,          // grpc 微服务端口

    pub metrics_port: u16,     // prometheus metrics port

    pub graceful_wait_time: u64, // 平滑退出等待时间, 单位 s

    pub app_debug: bool,

    pub aes_key: String,

    pub aes_iv: String,

}


// config read and init app config

pub static APP_CONFIG: Lazy<AppConfig> = Lazy::new(|| {

    let config_dir = std::env::var("QA_CONFIG_DIR").unwrap_or("./".to_string());

    let filename = Path::new(config_dir.as_str()).join("app.yaml");

    println!("filename: {:?}", filename);

    let c = Config::load(filename);

    // read config to struct

    let conf: AppConfig = serde_yaml::from_str(c.content()).unwrap();

    if conf.app_debug {

        println!("conf: {:?}", conf);

    }

    conf

});

```

从上述 app.rs 代码片段中看出, AppConfig 结构体的每个字段对应了 app.yaml 配置文件的每个区块的配置字段。APP\_CONFIG 是一个 static 静态变量, 通过 once\_cell 包提供 sync::Lazy 结构体提供的新方法, 它接收一个闭包作为参数。在这个闭包中, 首先使用 config 模块提供的 Config::load 函数读取了 app.yaml 文件内容。然后, 通过 serde\_yaml::from\_str 函数将读取到的内容反序列化到 AppConfig 结构体对应的 conf 变量中。由于 APP\_CONFIG 是一个全局静态变量, 因此在 crates/qa-svc/main.rs 文件中, 可以直接使用, 其代码片段如下:

```
info!("app_debug:{:?}", APP_CONFIG.app_debug);
info!("current process pid: {}", process::id());

// gRPC 微服务启动地址
let address: SocketAddr = format!("0.0.0.0:{}", APP_CONFIG.app_port).parse().unwrap();
println!("app run on: {}", address.to_string());
```

### E1.3 grpcurl 工具安装和基本使用

为了验证 qa-project/crates/qa-svc 服务是否正常运行, 接下来让我们安装 grpcurl 工具。grpcurl 是由 FullStory 的工程师开发的一个 grpc 命令行工具, 基于 curl 命令的拓展, 专门用于处理 gRPC 请求。通过 grpcurl 工具, 开发人员可以直接从终端向 gRPC 服务器发送请求, 并以 json 或 pb 格式接收响应, 对快速测试、调试或了解 gRPC 服务非常有用。

在 MacOS 系统下安装 grpcurl 工具非常简单, 只需要执行 brew install grpcurl 命令即可。假设你使用的是其它操作系统, 并且本地已安装了 Go 语言的工具链 go 工具, 那你可以直接运行如下命令, 就可以安装 grpcurl 工具。

```
go install github.com/fullstorydev/grpcurl/cmd/grpcurl@latest
```

当我们安装好 grpcurl 工具后, 执行如下命令, 就可以验证用户注册和登录功能。

```
grpcurl -d '{"username":"daheige","password":"123456"}' -plaintext 127.0.0.1:50051
qa.QAService.UserRegister
```

用户注册和登录效果如下图 E1-3-1 所示:

```
Finished `dev` profile [unoptimized + debuginfo] target(s) in 11.99s
Running `./Users/heige/web/rust/daheige/qa-project/target/debug/qa-svc`
Hello, qa-svc
filename:"./app.yaml"
conf:AppConfig { mysql_conf: MysqlConf { dsn: "mysql://root:root123456@127.0.0.1/qa_sys", max_connections: 100, min_connections: 10, max_lifetime: 1800, idle_timeout: 300, connect_timeout: 10 }, pulsar_conf: PulsarConf { addr: "pulsar://127.0.0.1:6650", token: "" }, redis_conf: RedisConf { dsn: "redis://:@127.0.0.1:6379/0", max_size: 300, min_idle: 3, max_lifetime: 1800, idle_timeout: 300, connection_timeout: 10, cluster_nodes: None }, app_port: 50051, metrics_port: 2338, graceful_wait_time: 3, app_debug: true, aes_key: "YiBX0z9WnJjsS5aNXmi0AeT1yTPZZJYa", aes_iv: "3ZQEpwP9DbK4h1Z0" }
app run on:0.0.0.0:50051
prometheus at:0.0.0.0:2338/metrics
current insert user id = 1
username:daheige
[]

~ (-zsh)

Last login: Sun Jul  7 21:54:48 on ttys011
You have new mail.
~ grpcurl -d '{"username":"daheige","password":"123456"}' -plaintext 127.0.0.1:50051 qa.QAService.UserRegister
{
  "state": "1"
}
~ grpcurl -d '{"username":"daheige","password":"123456"}' -plaintext 127.0.0.1:50051 qa.QAService.UserLogin
{
  "token": "EGzFr0MNLlzn0geFnGlg0s581G59VWQEA8Eqc8nyQLpc42gYgr/Eze1H5ZhD23xJD5suPZTr8KUvQ+okGGIjBb3+qxaLj+FSvDZArmz5oks="
}
~
```

图 E1-3-1 grpcurl 工具验证用户注册和登录

## E1.4 Logger 日志模块

为了在整个 QA 问答系统中更好地使用日志记录功能，我通过 Rust 第三方库 chrono、env\_logger 封装了 Logger 模块，具体实现代码如下所示：

// part11/qa-project/crates/infras/src/logger.rs 文件

```
use chrono::Local;
```

```
use std::io::Write;
```

```
// 自定义 Logger 结构体
```

```
pub struct Logger {
```

```
    is_custom: Option<bool>, // 日志初始化是否自定义
```

```
}
```

```
impl Logger {
```

```
    pub fn new() -> Self {
```

```
        Self { is_custom: None }
```

```
    }
```

```
    pub fn with_custom(mut self) -> Self {
```

```
        self.is_custom = Some(true);
```

```

        self
    }

// 初始化日志配置
pub fn init(&self) {
    if self.is_custom.is_none() {
        env_logger::init(); // 使用 eng_logger 默认方式初始化
        return;
    }

    // 自定义 env_logger env settings

    let env_config =
        env_logger::Env::default()
            .filter_or(env_logger::DEFAULT_FILTER_ENV, "debug");

    let mut builder = env_logger::Builder::from_env(env_config);

    builder
        .format(|buf, record| {
            let level = record.level();

            // 通过 default_styled_level 方法设置不同 level 日志颜色标识
            // let level = buf.default_level_style(level);

            writeln!(
                buf,
                "{} {} [{}:{}] {}",
                Local::now().format("%Y-%m-%d %H:%M:%S"), // 时间格式
                level, // 日志级别
                record.module_path().unwrap_or("unnamed"), // 模块名
                record.line().unwrap_or(0), // 行号
                &record.args() // 日志 message body 内容
            )
        })
    })
}

```

```

        .init();
    }
}

```

在上述代码中，首先引入了 `chrono::Local` 模块和 `std::io::Write trait`（特征）。其中 `chrono::Local` 主要用于日志记录时间格式的自定义，`Write trait` 主要用于 `env_logger` 库自定义日志写入格式。然后，定义了 `Logger` 结构体，`is_custom` 字段，是一个 `Option bool` 类型，用于 `env_logger` 记录日志是否自定义格式。接着，在 `impl` 块中为 `Logger` 添加了 `new`、`with_custom` 和 `init` 三个方法。

为了在 `qa-project/crates` 中的每个子项目中都能够使用 `infras/src/logger.rs` 模块，需要在 `infras/src/lib.rs` 中添加如下代码：

```
pub use logger::Logger; // 使用 pub use 重新导出 Logger 结构体
```

## E1.5 gRPC-Gateway 运行原理

gRPC-Gateway 主要作用在于它允许开发者同时提供 gRPC 客户端和 HTTP JSON 格式的 API，这对需要同时支持这两种通信协议的应用来说非常有用。此外，它还支持多种端实现方式，包含 gRPC、OpenAPI/Swagger 文档和自定义 HTTP 后端服务，这使得它成为一个灵活且强大的工具，能适应各种不同的应用场景开发。下图 E1-5-1 是 gRPC-Gateway 基本原理图：

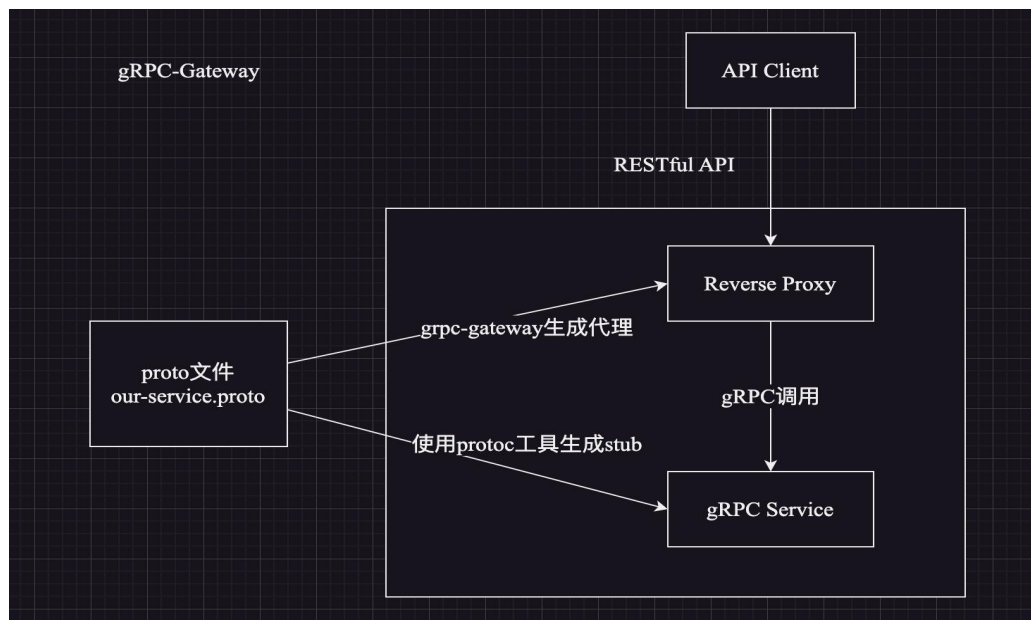


图 E1-5-1 gRPC-Gateway 运行方式

从图 E1-5-1 中可以看出，gRPC-Gateway 工作原理是基于 `google.api.http` 的注解实现的。这些注解在 gRPC 服务定义中指定了 HTTP 请求的方法、路径和请求/响应体与



gRPC 消息的映射关系。通过 protoc 工具可以生成对应的反向代理服务端和 gRPC stub 代码。当客户端发送一个 HTTP RESTful API 请求到反向代理服务时，反向代理服务会将请求的 JSON 对象转换为 pb Message 数据类型，然后调用下游的 gRPC 服务接口。通过这种转换方式使得 gRPC 服务能够以 HTTP 的形式对外提供服务，从而实现了 gRPC 与 HTTP/JSON API 的兼容。

## E1.6 prometheus 和 grafana 安装

以下内容是在 Linux Ubuntu 系统下安装 prometheus 服务和 grafana 工具的基本步骤，其它操作系统下的安装步骤，你可以参考官方手册。

### Linux Ubuntu 系统安装 prometheus 服务:

#### 1) 更新包索引

```
sudo apt-update
```

#### 2) 将 prometheus 官方仓库添加到 sources 中

```
echo "deb https://packages.prometheus.io/apt/ubuntu $(lsb_release -sc) main" | sudo tee  
/etc/apt/sources.list.d/prometheus-server.list
```

#### 3) 导入 prometheus 的 GPG 密钥和更新包索引

```
wget -qO- https://packages.prometheus.io/apt/doc/apt-key.gpg | sudo apt-key add -  
sudo apt-update
```

#### 4) 安装 prometheus 软件包

```
sudo apt-get install prometheus
```

当 prometheus 服务在安装完成后会自动启动，你可以通过以下命令查看运行状态:

```
sudo systemctl status prometheus
```

prometheus 服务默认配置放在/etc/prometheus/prometheus.yml 中，Web UI 界面默认将运行在 9090 端口，常用的命令如下:

```
# 重启 prometheus 服务
```

```
sudo systemctl restart prometheus
```

```
# 停止 prometheus 服务
```

```
sudo systemctl stop prometheus
```

```
# 启动 prometheus 服务
```

`sudo systemctl start prometheus`

当安装好 prometheus 服务后，你可以在浏览器中访问 `http://localhost:9090` 查看 prometheus 界面，如下图 E1-6-1 所示：

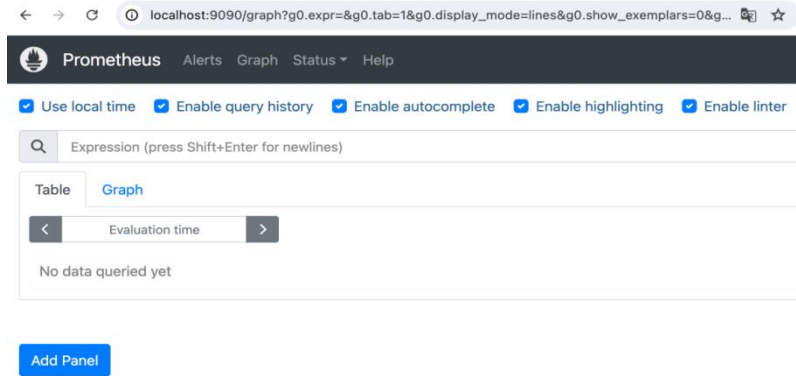


图 E1-6-1 prometheus web 界面

由于篇幅问题，我就不演示 Windows、MacOS 系统下如何安装 prometheus，你可以参考 prometheus 官网 <https://prometheus.io/docs/prometheus/latest/installation> 安装 prometheus。

#### Linux Ubuntu 系统安装 grafana 工具：

1) 使用 apt 更新包索引和导入 grafana 的 GPG 公钥

```
sudo apt-update
```

```
wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -
```

2) 将 grafana 的仓库添加到 apt 源列表中

```
echo "deb https://packages.grafana.com/oss/deb stable main" | sudo tee -a  
/etc/apt/sources.list.d/grafana.list
```

3) 再次更新索引包和安装 grafana 软件

```
sudo apt-update
```

```
sudo apt-get install grafana
```

4) 启动 grafana 服务并设置开机启动

```
sudo systemctl start grafana-server
```

```
sudo systemctl enable grafana-server
```

上述内容就是 Ubuntu 系统上安装 grafana 工具的基本步骤。如果你想重启或停止 grafana 服务，可以执行如下命令：

```
# 重启 grafana
```

```
sudo systemctl restart grafana-server
```

```
# 停止 grafana
```

```
sudo systemctl stop grafana-server
```

由于篇幅问题，这里就不演示 Windows、MacOS 系统下如何安装 grafana，你可以前往 grafana 官网 <https://grafana.com>，下载并安装 grafana 软件。

## E1.7 supervisor 工具安装

以下内容是在 Linux Centos 系统上安装 supervisor 基本步骤：

1) 通过包管理器安装

```
sudo yum update && sudo yum install epel-release
```

```
sudo yum install -y supervisor
```

2) 创建配置文件路径和生成默认配置

```
sudo mkdir -p /etc/supervisor/conf.d/
```

```
sudo echo_supervisord_conf > /etc/supervisord.conf
```

这里需要注意的一点：supervisor 默认配置文件在/etc/supervisord.conf 中，你可以直接编辑该文件或者在/etc/supervisor.d/目录中创建一个以.ini 结尾的配置文件。

3) 启动 supervisord 守护进程和设置开机自动启动

```
sudo systemctl start supervisord
```

```
sudo systemctl enable supervisord
```

以上内容就是 Centos 系统安装 supervisor 服务的基本操作。如果需要在其它操作系统上面安装 supervisor 工具，参考官方网站：<http://supervisord.org/>。除了上面提到的 supervisorctl 命令管理应用程序进程之外，你还可以通过浏览器访问 <http://ip:9001> 管理 supervisor 服务，前提是你需要将 supervisord.conf 文件中的[inet\_http\_server]修改为如下内容：

```
[inet_http_server]
```

```
port=0.0.0.0:9001 # 设置访问的 ip 和 port
```

以下命令是 supervisor 常用命令，你可以根据实际情况执行不同的命令。

```
# 查看 supervisor 运行状态
```

```
supervisorctl status
```

```
# 关闭所有服务
```

`supervisorctl shutdown`

# 启动某个进程

`supervisorctl start your-program`

# 重启某个进程

`supervisorctl restart your-program`

# 停止某个进程

`supervisorctl stop your-program`

# 停止全部进程（备注：start、restart、stop 操作不会载入新的配置文件）

`supervisorctl stop all`

# 停止原有所有进程并载入新的配置文件

`supervisorctl reload`

# 根据最新的配置文件，启动新配置或有改动的进程

`supervisor update`

# 查看 supervisor 命令帮助

`supervisor help`

## 附录 F: Rust 和 Go 语言对比

在如今快速发展的软件开发领域中，选择合适的编程语言对项目的成功至关重要。Rust 和 Go 是两门现代编程语言，它们都各自独特的优势和擅长的领域。接下来，我将从不同的角度分析这两门语言，包含语言特性、性能和效率、生态系统、适用场景以及社区支持。

### Rust 语言:

- 设计哲学: Rust 强调安全性、速度和并发，它是一种多范式编程语言，特别适合系统编程。
- 语言特性: Rust 具有内存安全（无垃圾回收）、所有权模型、类型系统、模式匹配、高效的错误处理、函数式编程等特性，以及强大的类型系统和借用检查机制提供了编译时的内存安全保证，使得 Rust 能够在高性能、高效率的项目中发挥语言自身的优势。
- 性能和效率: Rust 提供了无垃圾回收的内存安全保证，减少了运行时开销。同时，Rust 编译器优化和零成本抽象特性提供了接近 C/C++ 的性能。
- 工具链: Rust 语言强大的 Cargo 包管理和 rustup 工具链安装器，提供了项目构建、包管理和分发、依赖管理和测试工具等功能，让开发者能够快速构建和维护项目。
- 适用场景: Rust 在操作系统、系统编程、高性能项目、网络编程、游戏开发、嵌入式系统、数据分析、基础设施等领域表现出色，受到广泛开发者的喜爱。

### Go 语言:

- 设计哲学: Go 由 Google 开发，以简洁、高效和易读性著称。它是一种静态类似、编译型语言，具有优秀的并发支持。
- 语言特性: Go 具有并发模型（goroutine 和 channel）、垃圾回收、语法简单、自带丰富的标准库等特性，使得开发者在短时间内能够快速上手和使用。
- 效率和性能: Go 的 GMP 并发模型使得它在处理高并发任务表现非常出色，高效率。Go 语言设计自带运行时，但由于垃圾回收，程序可能会发生延迟。
- 工具链: Go 提供了全面的 go 工具，包含代码格式化 gofmt、go build、go test、godoc 文档生成工具、go mod 依赖管理。

- 适用场景：Go 在云原生、微服务架构、网络服务、分布式系统等领域，充分发挥了其高性能和高效率的优势，成为开发快速和部署应用的理想选择。

从上述各方面对比可知，Rust 和 Go 语言都是现代化、高效率、高性能的编程语言。这两门语言并没有哪个好，哪个不好，因为它们各自都有自己的优势和适用场景。选择哪一门语言取决于团队熟悉度、学习成本、维护成本、以及项目需求和性能要求等。同时，还需要了解每种语言自身特点，才能更好地发挥它们各自的优势，以解决软件开发过程中的各种痛点和难题。

无论你是被 Rust 内存安全、类型安全、高性能等特性所吸引，还是被 Go 简单易用、高性能、高效率等特性所吸引。这两门语言都提供了丰富的学习资源、活跃的社区、优秀的开源项目，以及成长和探索的激动人心的机会。我非常鼓励你可以尝试一下这两门语言，尝试不同的领域开发，并参与 Rust 和 Go 开源社区，感受一下这两门语言未来发展趋势和走向。同时，我也希望你拥抱不同语言的优势，并结合你自身情况和兴趣爱好去学习它们，并将其应用到实际项目中。这样，你将开启一个创新、充满活力的编程体验之旅！