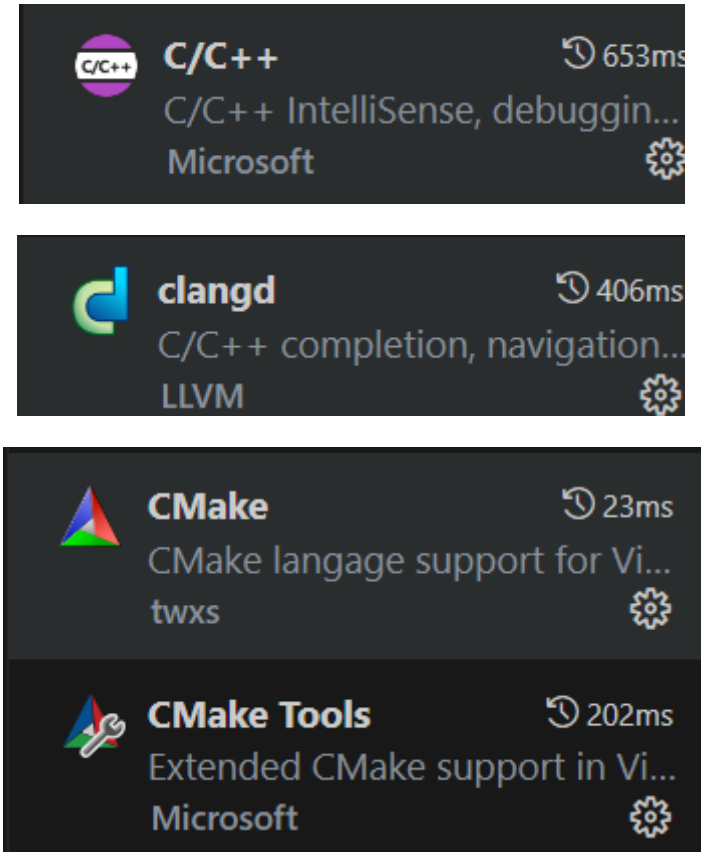
零声教育出品 Mark 老师 QQ: 2548898954

workflow 开发环境

vscode 插件安装



vscode 设置

> 在 *.vscode* 目录下，新建 *settings.json*

```
{
    "clangd.fallbackFlags": [
        "-I${workspaceFolder}/_include/workflow",
        "-I${workspaceFolder}/_include"
    ],
    "clangd.arguments": [
        "--background-index",
        "--compile-commands-dir=${workspaceFolder}/build.cmake/"
    ],
    "cmake.buildDirectory": "${workspaceFolder}/build.cmake",
    "cmake.buildEnvironment": {"CMAKE_EXPORT_COMPILE_COMMANDS": "ON"}
}
```

workflow 介绍

> 搜狗公司C++服务器引擎，编程范式。支撑搜狗几乎所有后端C++在线服务，包括所有搜索服务，云输入法，在线广告等，每日处理数百亿请求。这是一个设计轻盈优雅的企业级程序引擎，可以满足大多数后端与嵌入式开发需求。
>
> **特征：**
>
> - 快速搭建 http 服务器。
> - 可异步访问常见第三方服务：http，redis，mysql 和 kafka。
> - 构建异步任务流，支持常用的串并联，也支持更加复杂的DAG结构。
> - 作为并行计算工具使用。除了网络任务，我们也包含计算任务的调度。所有类型的任务都可以放入同一个流中。
> - 在`Linux`系统下作为文件异步IO工具使用，性能超过任何标准调用。磁盘IO也是一种任务。

> - 实现任何计算与通讯关系非常复杂的高性能高并发的后端服务。

workflow 编译安装

```
git clone https://github.com/sogou/workflow # From gitee: git clone https://gitee.com/sogou/workflow
cd workflow
make
cd tutorial
make
```

workflow 编程范式

程序 = 协议 + 算法 + 任务流

- 协议
  - 大多数情况下，用户使用的是内置的通用网络协议，例如http，redis或各种rpc。
  - 用户可以方便的自定义网络协议，只需提供序列化和反序列化函数，就可以定义出自己的client/server。
- 算法
  - 在我们的设计里，算法是与协议对称的概念。
    - 如果说协议的调用是rpc，算法的调用就是一次apc（Async Procedure Call）。
  - 我们提供了一些通用算法，例如sort，merge，psort，reduce，可以直接使用。
  - 与自定义协议相比，自定义算法的使用要常见得多。任何一次边界清晰的复杂计算，都应该包装成算法。
- 任务流

    - 任务流就是实际的业务逻辑，就是把开发好的协议与算法放在流程图里使用起来。
    - 典型的任务流是一个闭合的串并联图。复杂的业务逻辑，可能是一个非闭合的DAG。
    - 任务流图可以直接构建，也可以根据每一步的结果动态生成。所有任务都是异步执行的。

结构化并发与任务隐藏

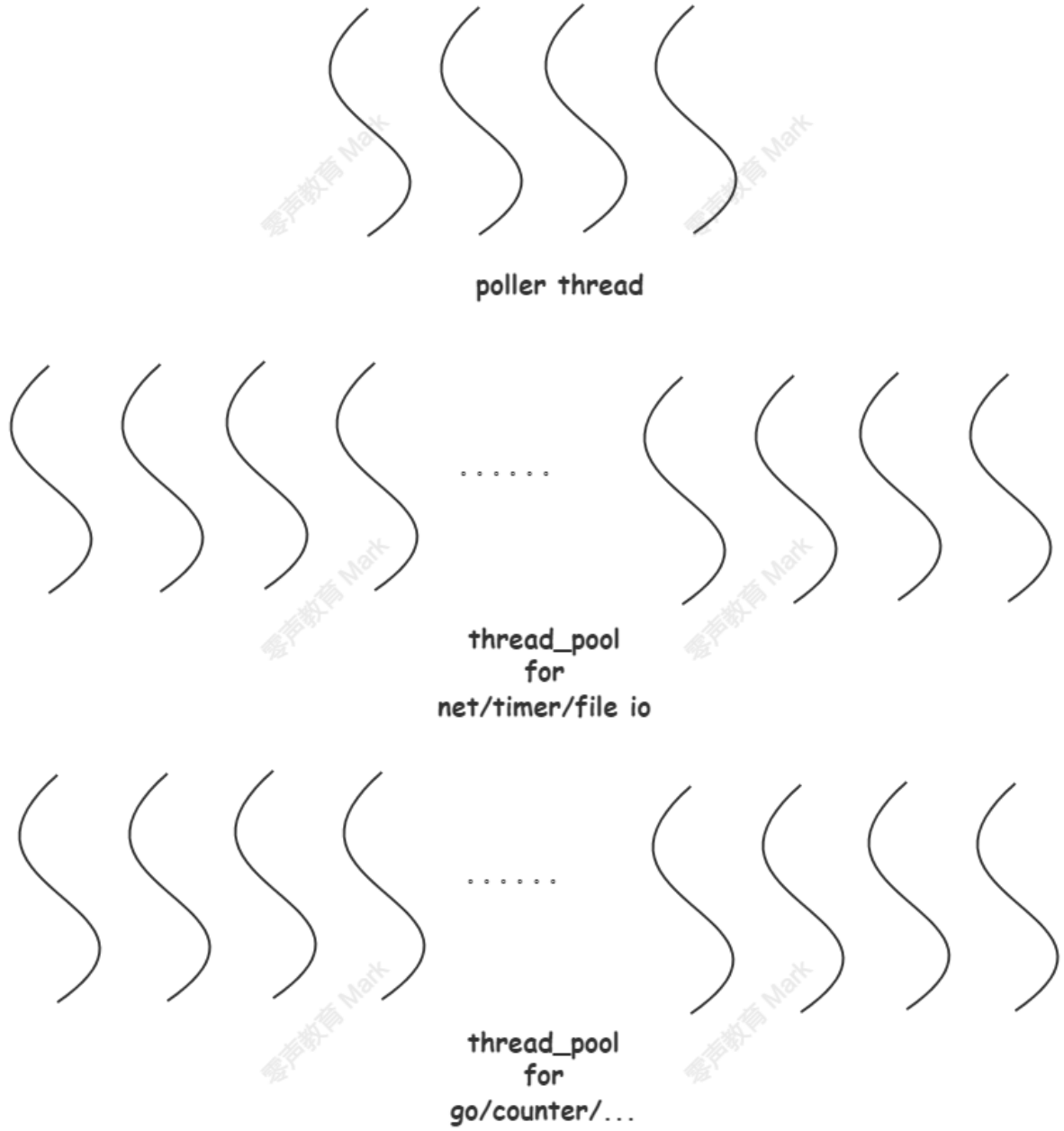
- 我们系统中包含五种基础任务：通讯，计算，文件IO，定时器，计数器。
- 一切任务都由任务工厂产生，用户通过调用接口组织并发结构。例如串联并联，DAG等。
- 大多数情况下，用户通过任务工厂产生的任务，都隐藏了多个异步过程，但用户并不感知。
    - 例如，一次http请求，可能包含许多次异步过程（DNS，重定向），但对用户来讲，就是一次通信任务。
    - 文件排序，看起来就是一个算法，但其实包括复杂的文件IO与CPU计算的交互过程。
    - 如果把业务逻辑想象成用设计好的电子元件搭建电路，那么每个电子元件内部可能又是一个复杂电路。
    - 任务隐藏机制大幅减少了用户需要创建的任务数量和回调深度。
- 任何任务都运行在某个串行流（series）里，共享series上下文，让异步任务之间数据传递变得简单。

回调与内存回收机制

- 一切调用都是异步执行，几乎不存在占着线程等待的操作。
- 显式的回调机制。用户清楚自己在写异步程序。
- **通过一套对象生命周期机制，大幅简化异步程序的内存管理**

    - 任何框架创建的任务，生命周期都是从创建到callback函数运行结束为止。没有泄漏风险。
        - 如果创建了任务之后不想运行，则需要通过dismiss()接口删除。
    - 任务中的数据，例如网络请求的resp，也会随着任务被回收。此时用户可通过`std::move()`把需要的数据移走。
    - 项目中不使用任何智能指针来管理内存。代码观感清新。
- 尽量避免用户级别派生，以`std::function`封装用户行为，包括：
    - 任何任务的callback。
    - 任何server的process。符合`FaaS`（Function as a Service）思想。
    - 一个算法的实现，简单来讲也是一个`std::function`。
    - 如果深入使用，又会发现一切皆可派生。

workflow 的线程模型



案例讲解

应用场景

• 高扇出场景

> 搜索服务一般都是高扇出场景；它需要调用许多下游模块多，考验网络通信框架的调度能力。
>
> 高扇出解释：一个节点与许多其他节点存在大量连接的情况。

> 高扇出的痛点：吞吐与长尾；提升吞吐需要提高单个请求的响应速度，所以需要尽量少切换网络收发线程，但是不切换容易导致处理的慢的资源堆积，长尾问题就很明显。

- **多 client 混用场景**

> 同时访问 redis/mysql/kafka 的数据管理需求。workflow 实现了常见的网络协议，不同协议的任务可以在底层调度层无缝打通。

- **SRPC**

> 业务层协议 IDL 使用了 protobuf，需要对里边定义的 service 支持简单的计算功能，同 时支持可以异步执行，如何能够快速搭建这样的服务。基于Workflow做底层调度的生态项目，拥有Workflow的网络性能优势，且自生成service接口，因此可以让 service 接口底层打通 Workflow 底层的异步 server 功能。除此之外使 用案例 2 中的其他协议或者计算任务都很方便。

- **嵌入式领域**
- **服务治理场景（服务发现）**
- **自定义协议接入**

> 场景：公网接入网需要 http 协议，但是后端服务是自定义私有协议，接入层需要自己开发。
>
> 常见的办法是使用 nginx，开发 ngx_module_t，但 nginx 的网络有 11 个阶段，内部资源纯自行管理，模块代码与框架代码完全耦合到一起，开发起来非常困难，出了问题也很难排查。

> workflow 可以派生基本网络层，实现消息的序列化/反序列化接口，即得到自定义协议的任务。然后启动 http server，创建自定义任务，再利用案例 4 中提到的转发功能，即可得到一个自定义协议接入层，转发性能无损耗。