产品需求

- 需求描述

> 设计一个爬虫程序，输入是一个`url`列表（种子`url`），爬虫下载`url`对应页面，保存在特定存档位置归档。爬虫分析页面内容，从里面提取新的`url`，继续爬取。
> 要求：重复`url`不必爬取。从种子`url`往下爬需要设置深度，最多只往下爬取`n`个页面即可。需要能控制并发度，需要能设定下载页面的超时和重试次数。
> 其它方面，可以自己发挥。例如，可以优雅退出程序。

- 其他要求
    1. 使用Golang语言
    2. 尽可能完善，考虑扩展性

技术设计分析

- 通过简要分析程序整体功能，可以明确需要至少以下子功能：网页下载、内容分析（提取url）、内容存储，此外还需要整体的控制逻辑用以串联以上功能，以下分别分析

网页下载

- 下载url对应的网页内容，根据需求，需要可以配置超时、重试次数等参数
- 功能非常明确，但是实现细节非常多，举例：客户端模拟、访问行为模拟、访问源控制（代理设置）等，需要将此子功能单独列为模块
- 明显是一个IO密集型任务（网络IO）
- 可以不持有状态（无状态），进行水平扩展

内容分析（提取url）

- 识别网页内容中的url并进行提取，用以提取下一层级的url再次进行爬取
- 功能非常明确，可考虑作为单独模块
- 明显是一个计算密集型任务
- 可以不持有状态（无状态），进行水平扩展

内容存储

- 需要存储的内容分为两个方面
    - url网页内容，宜使用文件系统存储，注意数据量扩展性问题
    - url元信息，用于记录是否已经爬取、爬取时间等状态信息，宜使用“数据库”概念进行封装，注意数据量扩展性问题
- 必须要明确内容存储子功能是一个“有状态”的功能，扩展性欠佳，必要时通过分片（sharding）方式进行扩展

整体控制逻辑

- 负责串联上述功能，包括检查重复爬取、限制爬取深度
- 可以不持有状态（无状态），进行水平扩展，可以通过与“内容存储”子功能交互模式来获取具体url的状态信息

技术设计

模块设计

- 请参看《整体逻辑视图.jpg》
    1. 需要爬取的url队列，不再解释
    2. 爬取任务worker池： 实现了网页下载功能的线程/协程池，从“需要爬取的url队列”中获取输入url，并将内容和url输出至“已爬取内容队列”
    3. 已爬取内容队列，不再解释
    4. 爬取内容分析池： 实现了分析、提取网页内容中的url功能，从“已爬取内容队列”中获取输入url以及内容，并将内容和url输出至“url/爬取内容/分析结果队列”
    5. url/爬取内容/分析结果队列， 不再解释
    6. 控制逻辑任务worker池，从“url/爬取内容/分析结果队列”提取结果，执行存储等操作，单个worker的功能请参看《控制逻辑任务.jpg》
       a. 负责文件操作，将网页内容保存至文件系统中
       b. 负责数据库操作，将网页元信息保存至格式化的数据库之中
       c. 提取新的待分析的url，并输出至“需要爬取的url队列”中

技术选型

- 本次实现使用golang语言，基础技术选型包括
    - 各个队列使用channel实现
    - 各个worker池使用gorouting+channel技术实现
    - 涉及独立功能的部分尽可能进行封装，并使用interface替换，包括
        - 爬取
        - 分析
        - channel消息的发送、接收
- 对于网页元数据存储当前考虑使用关系型数据库，本次实现使用postgresql，但是并不涉及特定于关系性数据的dml，可以替换为任意的、可靠的nosql甚至文件存储

其他考量

- 监控：（代码中暂未实现）
    - 当前使用prometheus的metric方式暴露以下指标
        - 各个worker完成的任务数量
        - 各个worker执行单次执行的任务耗时
    - 其他监控指标，例如队列长度可以实现，但是本次未写在代码中
    - 其余监控，例如文件系统大小、数据库总量暂不涉及
- 扩展性：（仅考虑水平扩展）
    - 当前仅涉及到存储的模块存在扩展性问题，包含两个方面
        - 文件系统存储，读写性能、文件数量、文件大小
        - 数据库存储，数据（表）总量，读写速度
        - 考虑使用域名（或hash后按字母）作为sharding key进行分片，但本次不涉及
    - 当规模进一步扩大时，可以将各个模块单独建立集群，并将channel的通讯方式修改为message broker（例如rabbitmq）或rpc调用进行通讯

数据库设计

- 存储网页元数据的数据库表设计如下，使用一张表，名为 `pages`， 字段如下（注意未做优化）

```
id 自增id
url url，注意移除了协议和hash tag部分
domain 从url中提取出的域名，不含端口号信息
state 状态， 0/新创建 1/爬取成功 2/爬取失败，当前不会针对已经下载好、或下载失败的页面再次进行下载，所以不必存储多份下载元信息数据
remark 描述信息，例如爬取错误描述
paths 一个json字符串，二维数组格式，保存了从seed url中本url的所有路径信息
    当且仅当该结构中所有path的长度>=n时，认为此网页中包含的url不再需要爬取（爬取终止）
    例如[["a.b.c", "d.e.f"],["g,h,i", "j.k.l"],]
sub_urls 一个json字符串，数组格式，保存了此网页下的所有的url（子url）例如["a.b.c", "e.d.f"]
fetched_at 网页内容下载时间
created_at 常规字段
updated_at 常规字段
```

- 可以看到目前时间最简单的方式来描述元数据，没有使用范式来约束数据库设计，这里可以持续优化

配置文件说明

- 配置文件中各项配置说明，最佳的参数配置可能需要通过测试得到

```
log: // 日志配置
  context: true // 打印日志时是否携带文件、行号信息（将影响性能）
  level: debug

core: // 核心参数配置
  url_queue_size: 10 // 需要下载的url队列长度
  page_info_queue_size: 10 // 需要分析的url及其内容的队列长度
  parsed_page_info_queue_size: 10 // 需要被controller处理（进行存储）的队列长度
  seed_file_path: "./seed.txt" // 种子文件位置
  retry_task_scan_period: 300 // 每隔多久运行一次发起重试任务
  task_timeout: 300 // 处于pending状态多久后认为可重试
  check_completed_period: 300 // 每个多久检查一次是否程序是否完成运行

database: // 数据库配置
  url: "postgres://crawler:123456@localhost:5432/crawler?sslmode=disable"

storage: // 网页内容存储位置
  location: "./pages"

downloader: // 下载设置
  worker: 3 // 并发度
  timeout: 5
  retry: 3

analayer: // 分析提取url的analyzer任务的并发度
  worker: 3

controller: // 执行存储任务的controller的并发度
```

```
  worker: 3
```

代码结构说明

- cmd目录包含了main函数
- config目录包含了一个示例配置文件，yaml格式
- docs中包含了相关文档说明
- src中为程序的逻辑源代码，区分如下
    - analyzer 为从网页内容中提取url的功能
    - config 为解析、提取配置文件内容的struct
    - controller实现了提取analyzer处理结果并与存储、数据库进行交互的逻辑功能
    - core中包含了启动seed任务、重试、判断整体任务是否结束的逻辑
    - dbstorage为数据库操作的逻辑封装
    - downloader为网页下载功能
    - entity为程序中在不同功能间传输信息用到的数据结构
    - enum为简单的变量定义
    - filestorage包含了文件系统操作的简单封装
    - routingpool实现了一个最为简单的协程池
        - 上文提到的analyzer、downloader和controller都可以水平扩展，故可以使用线程池方式进行承载
    - server为服务容器，在其中初始化程序的各个变量（包括channel）、启动协程池、启动各类定时任务
    - util为简单的辅助功能封装

关于存储的更多说明

- 可以看到在上述功能中特意将下载、分析、控制设置为可扩展，所有的逻辑控制操作（例如获取下一批需要爬取的url）都在存储侧完成，这势必导致系统瓶颈极易在存储侧发生
- 存储网页内容（文件存储）比较简单，只需要注意文件系统限制、注意分散文件夹即可
- 存储网页爬取过程的信息（即元信息/meta信息）比较困难，这里使用了关系型数据，并配合事务性操作来保证一定程度上的数据可靠性，相关描述如下：
    - 在爬取网页之前，必须在数据库中插入page记录，设置状态（state） 为pending
    - 在爬取网页并完成子url提取后，再查找数据库对应记录并进行更新
        - 如果记录已经处于成功状态，则说明此次为重复任务（正常逻辑不会出现此状态，但可能因为系统处理缓慢、重试机制导致），直接丢弃
            - 注意，这里将保持第一次爬取到的结果，包括内容和子url。但是后续可以升级成更新机制（例如距上次爬取24小时候后可再次爬取）
        - 如果记录状态处于pending，则直接保存信息，同时分析插入子url记录，并提交相关任务

        - 需要特别注意，在设置子url记录时，及时某个子url已经被爬取，但是因为爬取深度的关系，所以再次分析子url的记录（以及孙子url的记录），并根据深度再次启动相关任务
            - 代码中对于这一块的处理比较复杂，但是是按照上述流程执行的
    - 在判断程序是否执行完毕时依据的是数据库中是否存在pending的记录。对于单库来讲此过程极易实现，但是如果执行了sharding，则此功能实现将异常复杂甚至不可行，目前没有想到优化方案