MEP 20 -- Support RESTful API for Milvus

Current state: Under Discussion

ISSUE: https://github.com/milvus-io/milvus/issues/7700

PRs:

Keywords: Restful API

Released:

Summary(required)

Implement a http server on proxy module, serves http request and convert to same request type as grpc request.

Motivation(required)

Milvus 1.1 used to support restful API, we've also seen many other search engines such as elastic search and solr supports both http and rpc requests. It would be much easier for applications written in language such as PHP and rust to use http to utilize milvus.

Public Interfaces(optional)

(see https://github.com/milvus-io/milvus/blob/master/internal/proto/milvus.proto)

According to gRPC, the HTTP resource names can be:

- collections
- partitions
- aliases
- indexes
- records
- compactions
- ...

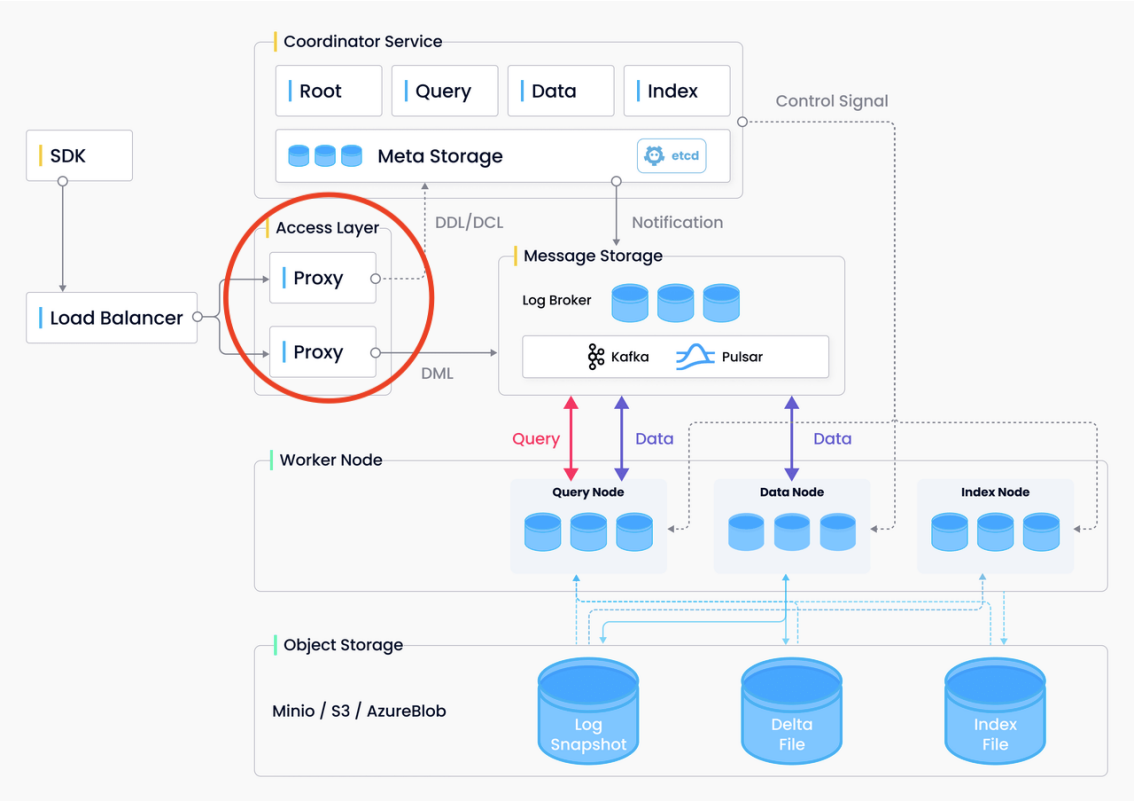
APIs: TODO: translate gRPC interface in RESTful API.

```
service MilvusService {
  // (omit api version prefix)
  // POST /collections
  rpc CreateCollection(CreateCollectionRequest) returns (common.Status) {}
  // DELETE /collections/:collectionName
  rpc DropCollection(DropCollectionRequest) returns (common.Status) {}
  // GET /collections/:collectionName
  rpc HasCollection(HasCollectionRequest) returns (BoolResponse) {}
  // PUT /collections/:collectionName/load
  rpc LoadCollection(LoadCollectionRequest) returns (common.Status) {}
  // DELETE /collections/:collectionName/load
  rpc ReleaseCollection(ReleaseCollectionRequest) returns (common.Status) {}
  // GET /collections/:collectionName/schema
  rpc DescribeCollection(DescribeCollectionRequest) returns (DescribeCollectionResponse) {}
  // GET /collections/:collectionName/statistics
  rpc GetCollectionStatistics(GetCollectionStatisticsRequest) returns (GetCollectionStatisticsResponse) {}
  // GET /collections-list
  rpc ShowCollections(ShowCollectionsRequest) returns (ShowCollectionsResponse) {}

  // POST /collections/:collection/partitions
  rpc CreatePartition(CreatePartitionRequest) returns (common.Status) {}
  // DELETE /collections/:collection/partitions/:partition
  rpc DropPartition(DropPartitionRequest) returns (common.Status) {}
  // GET /collections/:collection/partitions/:partition
  rpc HasPartition(HasPartitionRequest) returns (BoolResponse) {}
  // PUT /collections/:collection/partitions/:partition/load
  rpc LoadPartitions(LoadPartitionsRequest) returns (common.Status) {}
  // DELETE /collections/:collection/partitions/:partition/load
  rpc ReleasePartitions(ReleasePartitionsRequest) returns (common.Status) {}
  // GET /collections/:collection/partitions/:partition/statistics
  rpc GetPartitionStatistics(GetPartitionStatisticsRequest) returns (GetPartitionStatisticsResponse) {}
  // GET /collections/:collection/partitions-list
  rpc ShowPartitions(ShowPartitionsRequest) returns (ShowPartitionsResponse) {}

  // ...

}
```

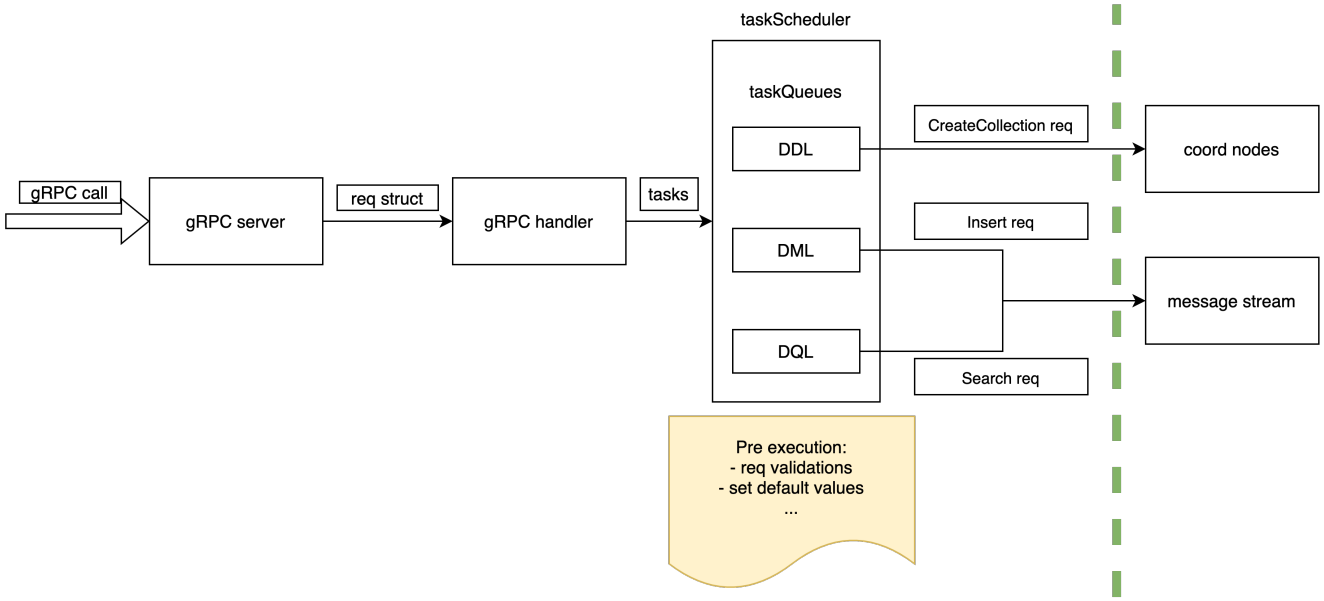
Design Details(required)

Web module

As we can see from the milvus v2 architecture, all the gRPC requests first go into the Access Layer, which is the milvus-proxy service, where the requests are then propagated to other funtional services of milvus. So that's where we should put our web server.



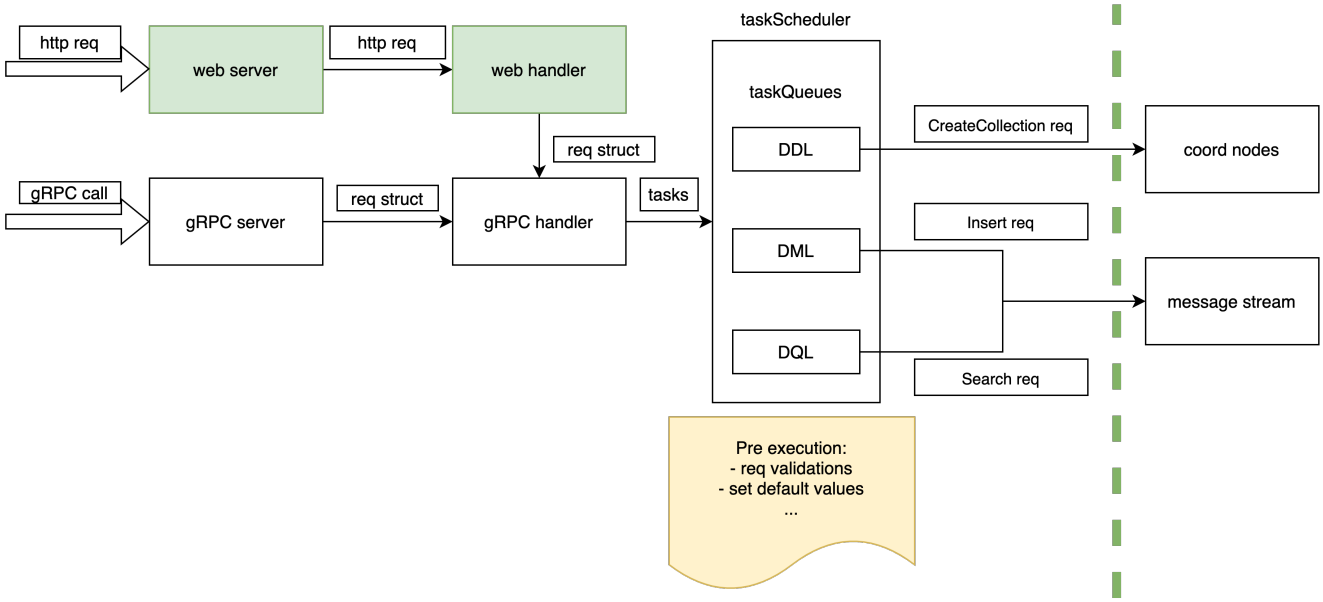
Let's take a close look inside the milvus-proxy



Above is the dataflow of milvus-proxy while receiving requests, modules left of the green dotted line is inside milvus-proxy

- The gRPC call received by gRPC server is decoded into a req struct which is defined in protobuf
- The corresponding method handler is called with the req struct, where the req body is wrapped as a task.
- The task is then put into taskScheduler's taskQueues by its request type as DDL, DML or DQL (we don't have DCL for now). For example, the CreateCollection task goes into the DML queue.
- The task is then dequeued by taskScheduler. Some "pre-execution" process will be done, such as request validations, assign some default values and etc.
- The request is then propagated to other components of milvus.

So the best way to reuse code is as below:



We integrate these green colored modules to milvus-proxy, so that it can handle http calls.

After the http request parsed, it's translated into the same req struct as we mentioned above, and the rest things can be done by the gRPC handlers which were already implemented.

Web Framework

We'll use golang for easy integration with current proxy. And we'll use GIN is the most popular and also easy to use golang web framework.

TODO Tasks:

Preparation:

Refine the design details.
Decide on which web server framework we should use.
Initialize the framework together with an example of API code.

Coding:

Implements the APIs listed in milvus.proto one by one: each of us first claims a part of all the APIs, and finish them. (38 APIs in all)
Convert gRPC method to RESTful API
Implement API handler
Write unit test

Testing:

Add integration tests.

Docs:

Provide samples for user, such as implementing a hello milvus with http client

Test Plan(required)

- unit test with the go `httptest` package: example
- Integratin Test with http client.

Rejected Alternatives(optional)

References(optional)