laravel - week-1

Boot

app 核心的业务逻辑文件

bootstrap 启动目录，会在 index.php的load一次，主要是用于实例化 Illuminate\Foundation\Application对象，

```
//1. 这里很重要。就是配置app当前的路径，就是为了后期的类加载器
$app = new Illuminate\Foundation\Application(
    $_ENV['APP_BASE_PATH'] ?? dirname(__DIR__)
);

// 2.注册一些东西
$app->singleton(
    Illuminate\Contracts\Http\Kernel::class,
    App\Http\Kernel::class
);

$app->singleton(
    Illuminate\Contracts\Console\Kernel::class,
    App\Console\Kernel::class
);

$app->singleton(
    Illuminate\Contracts\Debug\ExceptionHandler::class,
    App\Exceptions\Handler::class
);


return $app;
```

public目录： index.php

```
define('LARAVEL_START', microtime(true));

// 提前加载 ， 这个是composer生成的依赖 ， 如果没有这个依赖就会报错。
require __DIR__.'/../vendor/autoload.php';

//这个就是全部的核心，类似于Java的那个SpringApplication意义，这个是
\Illuminate\Foundation\Application, Kernel可以说是我们的配置类，具体可以看一下
\Illuminate\Foundation\Http\Kernel这个。通过继承可以重写一些属性和方法
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);
```

```
// 处理，主要的请求响应逻辑
$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
);

// 发送
$response->send();

// 关闭连接
$kernel->terminate($request, $response);
```

router

> `router/web.php` ， 可以修改配置的， 不一定放这里， 可以修改`RouteServiceProvider`改类注册我们的router； 配置路由方法比较简单，builder跟着走就行了；

6种请求方式

```
Route::get("/user/{msg}",function ($msg){
    echo $msg;
});
```

请求，匹配正着

```
Route::get("/user/{msg}",function ($msg){
    echo $msg;
})->where('msg','[1-9]+');
```

数据库

首先我们需要返回一个连接对象， 可以在config目录下配置你的数据库信息

```
<?php

use Illuminate\Support\Str;
// 这里的env（），其实是我们.env配置的信息
return [
    'default' => env('DB_CONNECTION', 'mysql'),
    'connections' => [
        'mysql' => [
            'driver' => 'mysql',
            'url' => env('DATABASE_URL'),
```

```
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3307'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'unix_socket' => env('DB_SOCKET', ''),
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'prefix_indexes' => true,
            'strict' => true,
            'engine' => null,
            'options' => extension_loaded('pdo_mysql') ? array_filter([
                PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
            ]) : [],
        ],

    ],
    'migrations' => 'migrations',
];
```

以上就是一个数据库的连接建立,首先你得建立一张表

先来个简单demo

```
class StudentController extends \Illuminate\Routing\Controller
{

    /**
     * @return array
     */
    public function index()
    {
        $users = DB::selectOne('select * from user where id = ?', [1]);
        return array('result'=>$users);
    }
}
```

设置路由访问就可以了。

然后我们开启mysql日志，看一下具体的情况；

```
// 查看日志是否记录
mysql> show variables like '%general%';
+------------------+----------------------------------+
| Variable_name    | Value                            |
+------------------+----------------------------------+
| general_log      | OFF                              |
| general_log_file | /var/lib/mysql/5279e5ef37cd.log |
+------------------+----------------------------------+

// 这里是开启日志记录
mysql> SET GLOBAL general_log = 'On';
Query OK, 0 rows affected (0.02 sec)
```

你会发现它每次请求都会：

```
2020-04-17T04:52:25.699397Z    32 Connect root@172.21.0.1 on laravel using
TCP/IP
2020-04-17T04:52:25.701706Z    32 Query use `laravel`
2020-04-17T04:52:25.703212Z    32 Prepare set names 'utf8mb4' collate
'utf8mb4_unicode_ci'
2020-04-17T04:52:25.704078Z    32 Execute set names 'utf8mb4' collate
'utf8mb4_unicode_ci'
2020-04-17T04:52:25.705260Z    32 Close stmt
2020-04-17T04:52:25.705707Z    32 Prepare set session
sql_mode='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION'
2020-04-17T04:52:25.706443Z    32 Execute set session
sql_mode='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION'
2020-04-17T04:52:25.707608Z    32 Close stmt
2020-04-17T04:52:25.708123Z    32 Prepare select * from user where id = ?
2020-04-17T04:52:25.709542Z    32 Execute select * from user where id = 1
2020-04-17T04:52:25.711240Z    32 Close stmt
2020-04-17T04:52:25.721942Z    32 Qui
```

连接 -> 查询 ->断开连接， 这样子效率很低， 如果建立长连接，会很快的，以为tcp的握手时很耗时的；

总结一句没有池化的概念， 慢；

Eloquent

> 这个是个啥呢， 类似一Java的Bean对象->SQL映射，mapping的含义，或者类似于JPA的entity；
> 既然是Mapping的概念，那么可以通过orm映射，很好的简化写sql；

```
class User extends Model
{
  // 告诉表的名称
    protected $table = "user";
}
```

其次就可以使用了

```
public function index()
{
// 这里不懂为啥可以直接 User::where('id',1)->get() 这么就可以执行
    return array('result'=>User::query()->where('id',1)->get());
}
```

PHP语言把，它没有静态与非静态方法的区别，唯一的区别就是非静态方法可以拿到一个this指针，而静态方法拿不到，但是非静态也可以拿静态来使用；

```
class Demos{
    // 非静态方法
    function invoke(){
        echo "invoke";
    }
}
// 这里可以执行，但是如果我们的invoke使用了$this就会报错
Demos::invoke();
```

因此也就是上诉的问题了；

Like 语法如下；

```
where('name','like','t%')
```

多个判断如下 , where 默认是一下几个参数，第一个表字段名，第二个值，第三个不知道，第四个就是连接符号，连接上一个语句的；orwhere=下面我写的这个；

```
where('id',1,null,'or')
```

还有一个get语句， 方法如下,columns是一个数组，我们将我们查询的字段返回， 也就是 select 和 from中间的部分

```
public function get($columns = ['*'])
```

还有些功能的比如 `orderby`,`desc`,`having`,聚合操作

```
$sum = User::query()->sum('id'); // 这个后面必须是字段名称，不能*或者1等
```

分页操作？

目前misc-api的有个接口分页逻辑不恰
当，`\App\Services\Peccancy\PeccancyService::listInfo`此实现的分页逻辑不合理，它是先查询出全部，然后我们的业务层面进行分页操作，也就是每次都会查询全部。

查看日志：

```
[2020-04-17 17:38:39] development.DEBUG: array (
  0 => 907512,
  1 => 2,
  2 => 1,
)
[2020-04-17 17:38:40] development.INFO: DB query {"sql":"select count(*) as aggregate from `report_peccancy` where `user_id` = ? and `is_answer_reduction` = ?","bindings":[907512,1],"time":41.05}
```

没有去开启线上的sql日志，所以这个可以发现确实如此。这个分页做法不合理

正确的做法，其实也没有，主要是业务逻辑是

```
$lengthAwarePaginator = User::query()->paginate(5);
```

这个执行逻辑是

```
2020-04-17T09:50:15.587624Z	   79 Close stmt
2020-04-17T09:50:15.588088Z	   79 Prepare select count(*) as aggregate from `user`
2020-04-17T09:50:15.589053Z	   79 Execute select count(*) as aggregate from `user`
2020-04-17T09:50:15.590218Z	   79 Close stmt
2020-04-17T09:50:15.590803Z	   79 Prepare select * from `user` limit 5 offset 5
2020-04-17T09:50:15.591638Z	   79 Execute select * from `user` limit 5 offset 5
2020-04-17T09:50:15.594208Z	   79 Close stmt
```

先聚合然后再次查询，因为框架没办法缓存；

Middleware

> 一个前置、后置的拦截器

```
use Closure;
```

```php
class PreInterceptor
{
    public function handle($request,Closure $next){
      // 1.前置
        $response = $next($request);
      // 2.后置
        return $response;
    }

  // 请求发出去之前，回掉
      public function terminate($request, $response)
    {
        // invoke
    }
}
```

基本就是这个逻辑， 可以通过`app/Http/Kernel.php` 这里添加进入，

有 全局， 组， 路由的

```php
Route::group(['middleware' => ['web']], function () {
    //路由组内
});
```

缓存

> Facade 是一个很重要的组件， 很多应用都是基于这个组件开发的； 例如cache也是

Facades

> 脸的意思， 也就是正面， 设计模式里有一个叫做外观模式，其实就是黑盒使用；

其实在PHP里面有个高级的用法，类似于Java的代理以及反射机制，当你这个类调用静态方法时，如果此时没有发现方法， 则会走`__callStatic`方法

```php
class Proxy
{
    public static function __callStatic($name, $arguments)
    {
        // 代理，只要是不是自己的方法都会走这里
        $clazz = new Clazz();
        // 真正去调用的
        return $clazz->$name(... $arguments);
```

```
    }

}
class Clazz
{
    static function insert()
    {
        echo "incoke\n";
    }
}
Proxy::insert();// 输出 invoke
```

这就是为啥我们看到一堆东西， 它并没有实现方法，缺可以调用

那么如何使用呢？

首先我们创建一个 `app/Facades/Real/MyDatabase.php`

```
class MyDatabase
{
    function getConnect(){
        return "connect";
    }
}
```

其次我们还要告诉容器，我们改对象需要映射到我们这里,在这里可以提供帮助 `\App\Facades\MyDatabase`

```
class MyDatabaseProxy extends Facade
{
    protected static function getFacadeAccessor()
    {
        return 'my-database';
    }
}
```

其次就是我们具体的业务了，用app注册一下，告诉一下；

```
$app->singleton('my-database',\App\Facades\Real\MyDatabase::class);
```

基本流程就是这些了；

此后就可以用 `MyDatabaseProxy` 调用真正的 `MyDatabase` 了， 基本就是这个流程； 这个可以屏蔽一些底层的细节；