**路由模型绑定**

我们在使用路由的时候一个很常见的使用场景就是根据资源 ID 查询资源信息：

Route::get('task/{id}', function ($id) {

$task = \App\Models\Task::findOrFail($id);

});

Laravel 提供了一个「路由模型绑定」功能来简化上述代码编写，通过路由模型绑定，我们只需要定义一个特殊约定的参数名（比如 {task}）来告知路由解析器需要从 Eloquent 记录中根据给定的资源 ID 去查询模型实例，并将查询结果作为参数传入而不是资源 ID。

有两种方式来实现路由模型绑定：隐式绑定和显式绑定。

**隐式绑定**

使用路由模型绑定最简单的方式就是将路由参数命名为可以唯一标识对应资源模型的字符串（比如 $task 而非 $id），然后在闭包函数或控制器方法中对该参数进行类型提示，此处参数名需要和路由中的参数名保持一致：

Route::get('task/{task}', function (\App\Models\Task $task) {

dd($task); // 打印 $task 明细

});

这样就避免了我们传入 $id 后再进行查询，而是把这种模板式代码交由 Laravel 框架底层去实现。

由于路由参数（{task}）和方法参数（$task）一样，并且我们约定了 $task 类型为 \App\Models\Task，Laravel 就会判定这是一个路由模型绑定，每次访问这个路由时，应用会将传入参数值赋值给 {task}，然后默认以参数值作为资源 ID 在底层通过 Eloquent 查询获取对应模型实例，并将结果传递到闭包函数或控制器方法中。

路由模型绑定默认将传入 {task} 参数值作为模型主键 ID 进行 Eloquent 查询，你也可以自定义查询字段，这可以通过在模型类中重写 getRouteKeyName() 来实现：

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Task extends Model

{

public function getRouteKeyName() {

return 'name'; // 以任务名称作为路由模型绑定查询字段

}

}

以上就是隐式路由模型绑定的实现，是不是很简单？

**显式绑定**

显式绑定需要手动配置路由模型绑定，通常需要在 App\Providers\RouteServiceProvider 的 boot() 方法中新增如下这段配置代码：

public function boot()

{

// 显式路由模型绑定

Route::model('task\_model', Task::class);

parent::boot();

}

编写完这段代码后，以后每次访问包含 {task\_model} 参数的路由时，路由解析器都会从请求 URL 中解析出模型 ID ，然后从对应模型类 Task 中获取相应的模型实例并传递给闭包函数或控制器方法：

Route::get('task/model/{task\_model}', function (\App\Models\Task $task) {

dd($task);

});

注：如果路由模型绑定对应匹配记录不存在，将自动返回 404 响应。

由于在正式开发中，出于性能的考虑通常会对模型数据进行缓存，此外在很多情况下，需要关联查询才能得到我们需要的结果，所以并不建议过多使用这种路由模型绑定。

**兜底路由**

在 Laravel 5.6 中，引入了兜底路由功能。所谓兜底路由，就是当路由文件中定义的所有路由都无法匹配用户请求的 URL 时，用来处理用户请求的路由，在此之前，Laravel 都会通过异常处理器为这种请求返回 404 响应，使用兜底路由的好处是我们可以对这类请求进行统计并进行一些自定义的操作，比如重定向，或者一些友好的提示什么的，兜底路由可以通过 Route::fallback 来定义：

Route::fallback(function () {

return '我是最后的屏障';

});

这样，当我们访问一些不存在的路由，比如 http://blog.test/test/111，就会执行兜底路由中的处理逻辑，而不是返回 404 响应了。

**频率限制**

在 Laravel 5.6 中，还引入了频率限制功能。所谓频率限制，指的是在指定时间单个用户对某个路由的访问次数限制，该功能有两个使用场景，一个是在某些需要验证/认证的页面限制用户失败尝试次数，提高系统的安全性，另一个是避免非正常用户（比如爬虫）对路由的过度频繁访问，从而提高系统的可用性，此外，在流量高峰期还可以借助此功能进行有效的限流。

在 Laravel 中该功能通过内置的 throttle 中间件来实现，该中间件接收两个参数，第一个是次数上限，第二个是指定时间段（单位：分钟）：

Route::middleware('throttle:60,1')->group(function () {

Route::get('/user', function () {

//

});

});

以上路由的含义是一分钟能只能访问路由分组的内路由（如 /user）60 次，超过此限制会返回 429 状态码并提示请求过于频繁。

如果你觉得这种静态设置频率的方式不够灵活，还可以通过模型属性来动态设置频率，例如，我们可以为上述通过 throttle 中间件进行分组的路由涉及到的模型类定义一个 rate\_limit 属性，然后这样来动态定义这个路由：

Route::middleware('throttle:rate\_limit,1')->group(function () {

Route::get('/user', function () {

// 在 User 模型中设置自定义的 rate\_limit 属性值

});

Route::get('/post', function () {

// 在 Post 模型中设置自定义的 rate\_limit 属性值

});

});

这样，我们就可以通过为不同的模型类设置不同的 rate\_limit 属性值来达到动态设置频率限制的效果了。

**路由缓存**

使用路由缓存之前，需要知晓路由缓存只能用于控制器路由，不能用于闭包路由，如果路由定义中包含闭包路由将无法进行路由缓存，只有将所有路由定义转化为控制器路由或资源路由后才能执行路由缓存命令：

php artisan route:cache

如果想要删除路由缓存，可以运行：

php artisan route:clear

路由缓存对系统性能的提升应该是微乎其微的，但如果你很在意那几毫秒，则可以考虑，但是需要付出的代价是不能使用任何闭包路由，此外，由于使用路由缓存需要在每次变动路由后重新生成缓存，所以建议在应用部署脚本中执行 php artisan route:cache（运行此命令之前先要清理之前的缓存），即只在生产环境中使用路由缓存，本地开发环境路由经常变动，且没有性能方面的考虑，无需缓存。