**benchmark django rest framework各功能详解**

1. **引言**

在之前的一篇文章中（<https://github.com/hqsh/benchmark_django_rest_framework_demo/blob/master/Introduction/Introduction.md>），有对benchmark django rest framework（以下简称benchmark）的基本功能的介绍，本文进一步地讲解benchmark各种其他功能，这些功能在实际项目开发中得到较多的应用。灵活使用这些功能，相对使用常规的django rest framework（以下简称DRF）开发，能很大程度地减少开发的代码量，多数接口往往能减少一半、甚至更多的代码量。更因为代码量的减少，也能相应地减少程序的bug，以及开发后的维护成本。

1. **benchmark各功能配置项一览**

benchmark有如下两种类型的配置项：全局和单接口配置项，通过定义这些配置，即能实现很大一部分接口相关的功能开发。

1. 全局配置项：benchmark有类似django settings.py的配置文件，文件名是benchmark\_settings.py，里面的配置涵盖了和benchmark有关的全局的配置项（全大写字母命名的全局变量）。通过在django的settings.py文件中定义BENCHMARK\_SETTINGS = 'PATH.benchmark\_settings'（其中PATH代表benchmark\_settings.py文件的路径）来加载。这些配置是对所有继承自BenchmarkApiView的接口生效的。
2. 单接口的配置项：benchmark的单接口配置项是全小写字母命名的接口类变量或接口实例变量，这些配置是只对单个接口（类变量），或接口的某一次请求（实例变量）生效的。

如果同时存在多种类型的同一个配置项，配置项的优先级从高到低依次为：接口实例变量配置项、接口类变量配置项、全局变量配置项。

下表所列的是目前全部的配置项。在使用benchmark进行接口开发的代码中，如果同时存在全局和接口配置项，全局配置项名（全大写）和接口配置项名（全小写）仅大小写区别。为了让使用者能更快地上手使用，在表格的最右边一列，标注了配置项的配置频率，可以先熟悉配置频率高的。配置频率表示该配置需要设置或修改的频率，对于“低”频率的配置项，一般不需要修改或设置，仅在某些特定场景时会用到（“低”在这里仅表示配置项随平台、接口变化而变化的可能性，并不表示对应功能的使用频率，比如下面的CONVERT\_KEYS，按各个开发团队的习惯，一般各个应用平台的开发，都固定设置为True或False，不会经常变，但该配置在每个接口请求都会用到）。“需谨慎设置”表示那些全局配置项虽然可以设，但需要小心各个接口都会受影响，并且实际应用中更多的情况是只在接口中设置。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 配置项作用域 | 配置项名 | 全局配置项 | 接口配置项 | 默认值 | 配置项含义  （更详细的描述见本文后续内容，或相应的代码注释） | 配置频率 |
| 仅全局 | CODE | 必须设 | 无 | 无 | 接口请求返回的json字段中的状态码字段名。 | 低 |
| 仅全局 | SUCCESS\_CODE | 必须设 | 无 | 无 | 接口请求返回的json字段中的正确状态码字段值。 | 低 |
| 仅全局 | CODE\_OFFSET | 必须设 | 无 | 0 | 对benchmark框架内部定义的错误状态码（默认为1-199之间）进行的偏移（在平台自定义的错误状态码和默认的1-199有冲突时候，可以设置该值，内部错误码回往后挪）。 | 低 |
| 仅全局 | DICT\_RESPONSE\_BY\_CODE | 不建议修改 | 无 | 见代码，略 | 各种benchmark内部定义的错误状态码，一般不需要修改，也不建议修改。 | 低 |
| 仅全局 | DICT\_CUSTOM\_RESPONSE\_BY\_CODE | 可设 | 无 | {} | 平台自定义的状态码，需要注意不能与benchmark定义的1-199冲突，如果冲突，使用CODE\_OFFSET，将benchmark的状态码进行偏移。 | 高 |
| 仅全局 | PARENT\_VIEW | 可设 | 无 | 'ModelViewSet' | BenchmarkApiView继承的父类view，支持"APIView", "GenericAPIView", "ViewSet"或"ModelViewSet"，其中对swagger的兼容性最好的是"ModelViewSet"，默认为"ModelViewSet"。 | 低 |
| 仅全局 | LIMIT | 可设 | 无 | 'limit' | get请求分页查询时的请求参数名，表示一页最多多少条记录。 | 低 |
| 仅全局 | PAGE | 可设 | 无 | 'page' | get请求分页查询时的请求参数名，表示页码。 | 低 |
| 仅全局 | ORDER\_BY | 可设 | 无 | 'order\_by' | get请求查询结果中的排序方式请求参数名，只支持primary model及其外键的第一层字段排序，不支持按primary model一对多、多对多关联的其他表的字段排序。 | 中 |
| 仅全局 | Q | 可设 | 无 | 'Q' | get请求对应django Q object的或查询的Q函数对应的请求参数名，详细使用方法见benchmark\_settings.py中的注释。 | 低 |
| 仅全局 | Q\_OR | 可设 | 无 | '|' | get请求对应django Q object的或查询的配合Q函数的或关系运算符请求参数名，详细使用方法见benchmark\_settings.py中的注释。 | 低 |
| 仅全局 | Q\_AND | 可设 | 无 | '$' | get请求对应django Q object的或查询的配合Q函数的或关系运算符请求参数名，详细使用方法见benchmark\_settings.py中的注释。 | 低 |
| 仅全局 | FILE | 可设 | 无 | 'file' | 上传文件在请求体中的参数名，接口中的self.file即为该文件。 | 低 |
| 仅全局 | MODEL\_CREATE\_TIME | 可设 | 无 | None | model中定义的自动写入的创建时间字段名，设置该值能令post、put请求参数如果带上这些参数，直接忽视过滤掉，避免操作数据库的创建时间字段。 | 低 |
| 仅全局 | MODEL\_UPDATE\_TIME | 可设 | 无 | None | model中定义的自动写入的更新时间字段名，设置该值能令post、put请求参数如果带上这些参数，直接忽视过滤掉，避免操作数据库的更新时间字段。 | 低 |
| 仅全局 | MODEL\_CREATOR | 可设 | 无 | None | model中定义的创建者字段名，设置该值能令框架自动识别和写入post请求的用户（支持model对应字段为user外键或username字符串）到数据库中；同时也令接口请求权限配置（access）对put、delete请求支持对创建者有权限的访问控制（比如access={'put': 'creator', 'delete': 'creator'}，即可令该接口只允许创建者才能修改或删除primary model的数据）。 | 中 |
| 仅全局 | MODEL\_MODIFIER | 可设 | 无 | None | model中定义的修改者字段名，设置该值能令框架自动识别和写入put请求的用户（支持model对应字段为user外键或username字符串）到数据库中。 | 中 |
| 仅全局 | MODEL\_DELETE\_FLAG | 可设 | 无 | None | 如果有model使用标记位删除（即False或0表示未删除，True或非0表示记录被删除），设置该值为model中标记位删除字段的名称（字符串），即可令model的各种操作同django本身删除操作那样（框架支持各种外键等关联删除或标记位删除），框架能考虑到各种增、删、改、查的场景，方便开发者，其操作方式和接口返回，同普通没有标记位的常规方式。 | 低 |
| 仅全局 | DICT\_MODEL\_UNIQUE\_TOGETHER | 可设 | 无 | {} | 当设置过MODEL\_DELETE\_FLAG时，有标记位删除的model的单字段或多字段唯一性约束（unique=True或unique\_together）就不能正常使用了（因删除标记位会导致唯一性约束失效），benchmark框架支持通过在这个配置项中设置多字段的唯一性约束，在处理post、put请求时候，控制唯一性。 | 低 |
| 仅全局 | MANY\_TO\_MANY\_RELATION\_MODEL\_NAMES | 可设 | 无 | [] | 当设置过MODEL\_DELETE\_FLAG时，如果需要对多对多关系表也使用标记位删除，则不能用django自动生成并维护的多对多关系表，需要自己建一个多对多关系表，里面4个字段，自动id、到另外2个表的外键、和删除标记位。在这个数组中，填入这些自建的多对多关系表名，benchmark框架能支持这些多对多关系表的标记位删除。 | 低 |
| 仅全局 | MODEL\_JSON\_FIELD\_NAMES | 可设 | 无 | [] | 可以设置model中的json字段名，当post、put请求写数据库前，可将请求参数众多 json转换为字符串存储入库，当get请求读取数据库中的这些json字段时，会将数据库中的字符串转换为json。如果2种转换因json格式错误导致转换有错，则都不转换，效果和没有定义这个参数值一样，是个数据库中的普通字符串。  例：5.6节中的employee\_info在数据库中实际为json字符串，只需配置 MODEL\_JSON\_FIELD\_NAMES=('employee\_info', )，即可返回json。 | 中 |
| 仅全局 | OMIT\_UN\_EDITABLE\_FIELDS | 可设 | 无 | False | 设置get请求查询数据库的结果中是否忽略不可编辑的字段。比如model中有自动生成的创建时间、修改时间字段，使用django内置函数model\_to\_dict是无法将这些字段转换到字典中的，这样get请求就无法查询到记录的创建时间、修改时间。BenchmarkModel中重新定义了这个model\_to\_dict，并通过该变量，控制是否支持对这些不可编辑字段的读取。 | 低 |
| 仅全局 | CONVERT\_KEYS | 可设 | 无 | False | 接口请求是否需要驼峰、下划线转换。如果设置为True，接口请求、返回的参数名都为驼峰形式，在view中操作的参数名都为下划线形式，这些驼峰转下划线、下划线转驼峰都交给benchmark处理，开发接口时只需使用下划线风格的即可。 | 低 |
| 仅全局 | NEED\_AUTHENTICATION\_IN\_DEBUG\_MODE | 可设 | 无 | True | 设置该字段为False，可以令django的settings.py中的DEBUG=TRUE时，不需要认证，即对access定义的访问权限非None（非不可访问的method），不校验身份都能访问（相当于将权限视为'all'）。用于方便接口调试。 | 低 |
| 仅全局 | USER\_RIGHT\_AUTHENTICATE\_FUNCTION | 可设 | 无 | None | 可以设置为平台特有的认证方式的认证函数（非框架默认的django支持的各种方式），然后在接口的access变量中使用。  该函数有如下一些机制和要求：   1. 该函数是被benchmark框架内部用于权限认证时候调用使用的； 2. 需要是普通函数，或者类的静态函数，或者类的类函数； 3. 函数的输入参数依次为：BenchmarkApiView的实例和用户身份（对应access字典对应的value，并且value如果是字符串，benchmark框架会将字符串转换为字符串数组）；   函数的返回值可以是和DRF类似的True或False表示认证通过与否；也可以是benchmark框架的get\_response\_by\_code函数的返回值，benchmark框架会将该结果作为接口的返回，可用于在该函数中定义整个接口的返回信息（比如返回没有该接口的权限的具体原因）。 | 中 |
| 仅接口 | primary\_model | 无 | 可设 | None | 接口的主model。 | 高 |
| 仅接口 | using | 无 | 可设 | 'default' | 主model所在的数据库名。 | 高 |
| 仅接口 | access | 无 | 建议设 | {'get': 'all', 'post': 'all', 'put': 'all', 'delete': 'all'} | 接口查、增、改、删4个method的访问权限，若不设置则为4个method都可访问；若设置部分，则剩余部分的method都为不可访问（即值为None）。也可以设置为字符串数组，以支持多种身份的用户访问，如{'get': ['staff', 'creator']}。 | 高 |
| 仅接口 | select\_related | 无 | 可设 | None | 如果get请求要查询多个model，该参数值可以为一个字符串数组，数组内的字符串为从主model出发的django select\_related语法的字段名，同时也支持一对多、多对多关系。 | 高 |
| 仅接口 | values | 无 | 可设 | None | 接口响应的数据字段值筛选，支持请求url中带入参数，只支持primary model及其外键的第一层筛选，不支持primary model一对多、多对多关系的表的字段过滤。 | 中 |
| 仅接口 | values\_fields | 无 | 可设 | {} | 接口响应的数据字段值筛选，不支持url中带入参数，支持和primary model关联的各种多表关系的各个字段过滤。 | 高 |
| 仅接口 | values\_white\_list | 无 | 可设 | True | values\_fields是否白名单。 | 高 |
| 仅接口 | get\_one | 无 | 可设 | None | get请求是否只会查询到最多一个数据，默认情况下，uri中有pk变量（主键），则表示查询单条数据，这样的话，响应中的数据字段为字典，否则为数组。而有些接口需要返回字典、或数组，则通过该项配置可以将原本数组设为字典（设为True），或原本为字典的设置为数组（设为False）。 | 中 |
| 仅接口 | methods\_not\_need\_convert\_keys\_for\_request | 无 | 可设 | None | 当CONVERT\_KEYS=True时，定义此变量可配置请求参数不从驼峰转换为下划线的http method，比如值为['get ', 'post ']，则get和post请求的self.params，self.uri\_params，self.data中的参数名都不会被转换。 | 中 |
| 仅接口 | methods\_not\_need\_convert\_keys\_for\_response | 无 | 可设 | None | 当CONVERT\_KEYS=True时，定义此变量可配置响应参数不从下划线转换为驼峰的http method，比如值为['get ', 'post ']，则get和post请求的响应中的参数名都不会被转换。 | 中 |
| 仅接口 | enabled\_select\_related\_in\_params | 无 | 可设 | [] | 当接口的 enable\_select\_related\_in\_params为True时，具体那些select\_related参数是可以接受的，需要在这个变量里面配置。 | 低 |
| 仅接口 | bool\_params | 无 | 可设 | [] | 当接口的get请求有布尔类型值的请求参数时，设置该变量为这些参数名的数组，即可实现将参数值true、True、1转换为True，false、False、0转换为False。 | 中 |
| 仅接口 | additional\_data | 无 | 可设 | None | 有时候，如果在接口请求返回的多个数据中，有额外的字段（该字段和多个数据是一种一对多的关系，并且在多个数据的外面一层），使用return self.get\_response\_by\_code(data=data)等方法返回，是没法设置这个字段的，通过该变量，可以进行设置。  如5.9节中的代码实例，additional\_data是个字典类型的变量，接口返回如下，averageCreatedCount字段及其值是additional\_data中定义的：  {  **"status"**: 200,  **"msg"**: **"成功"**,  **"data"**: {  **"total"**: 2,  **"averageCreatedCount"**: 11.357142857142858,  **"results"**: [  {  **"accumulatedUnsolvedCount"**: 62,  **"createdCount"**: 0,  **"date"**: **"2017-09-14"**,  **"accumulatedCreatedCount"**: 307,  **"accumulatedSolvedCount"**: 245,  **"solvedCount"**: 1  },  {  **"accumulatedUnsolvedCount"**: 64,  **"createdCount"**: 5,  **"date"**: **"2017-09-15"**,  **"accumulatedCreatedCount"**: 312,  **"accumulatedSolvedCount"**: 248,  **"solvedCount"**: 3  }  ]  } } | 低 |
| 全局/接口 | DATA\_STYLE  data\_style | 可设 | 可设 | 'dict' | 接口返回字段中data字段的格式，默认值为最新质量管理部所使用的那种格式。 | 低 |
| 全局/接口 | OMIT\_UNDERLINES  omit\_underlines | 可设 | 可设 | True | 接口返回响应前，将下划线转驼峰，是否视连续的多个下划线为一个下划线。 | 低 |
| 全局/接口 | HTTP\_GET\_CHECK\_PARAMS  http\_get\_check\_params | 可设 | 可设 | False | get请求的参数是否用序列化进行检查。 | 低 |
| 全局/接口 | RENAME\_PARAMS  rename\_params | 需谨慎设 | 可设 | {} | 重命名get请求参数的参数名，全局配置需谨慎设，会影响所有接口低请求参数。 | 高 |
| 全局/接口 | RENAME\_URI\_PARAMS  rename\_uri\_params | 需谨慎设 | 可设 | {} | 重命名get请求uri中的参数的参数名，全局配置需谨慎设，会影响所有接口低请求参数。 | 低 |
| 全局/接口 | RENAME\_DATA  rename\_data | 需谨慎设 | 可设 | {} | 重命名post、put、delete请求体中的参数名，全局配置需谨慎设，会影响所有接口低请求参数。 | 高 |
| 全局/接口 | RENAME\_FIEDLS  rename\_fields | 需谨慎设 | 可设 | {} | 重命名接口响应中的数据字段里面的参数名，全局配置需谨慎设，会影响所有接口低请求参数。 | 高 |
| 全局/接口 | ENABLE\_SELECT\_RELATED\_IN\_PARAMS  enable\_select\_related\_in\_params | 需谨慎设 | 需谨慎设 | False | select\_related是否可在get请求的url中的参数中传递进接口。为了安全考虑，需要谨慎设置，因为可能让没有权限查询某些其他model的请求者通过该功能查询到这些model。 | 低 |
| 全局/接口 | REDIS\_IP  redis\_ip | 可设 | 可设 | 'localhost' | redis配置，ip地址。 | 低 |
| 全局/接口 | REDIS\_PORT  redis\_port | 可设 | 可设 | 6379 | redis配置，端口号。 | 低 |
| 全局/接口 | REDIS\_DB  redis\_db | 可设 | 可设 | 0 | redis配置，db号，最好别和celery等其他redis应用重复，以免有冲突问题。 | 低 |

在第4章中，将进一步以实例介绍其中的一些比较重要的或略复杂的配置。

1. **benchmark常用函数一览**

本章介绍常用的BenchmarkApiView和BenchmarkModel的各种常用函数，虽然BenchmarkApiView和BenchmarkModel实际有更多的函数，但在开发中，使用的较多的也就如下这些函数。这些函数是用于对接口的具体业务逻辑的处理。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 函数所在类 | 函数类型 | 函数名和参数 | 使用方式 | 函数说明 | 使用频率 |
| BenchmarkApiView | 实例函数 | get\_model(self) | 重定义该函数或调用父类同名函数，benchmark框架在处理get请求时候调用该函数 | 处理get请求业务逻辑的函数。如果接口定义了primary\_model，而接口没有重定义该函数，则get请求会对primary\_model进行新增数据操作，接口请求字段和model中的字段一一对应。 | 高 |
| BenchmarkApiView | 实例函数 | get\_redis(self) | 定义该函数，benchmark框架在处理get请求时候调用该函数 | 如果接口定义了get\_redis函数，一方面接口会按redis的配置（redis\_ip/redis\_port/redis\_db）维护redis连接；另一方面，get请求会调用get\_redis函数，而不是get\_model函数，在get\_redis函数中调用get\_redis\_value函数能方便地获取到redis的数据。 | 低 |
| BenchmarkApiView | 实例函数 | get\_redis\_value(self, key, data\_type='json', return\_with\_success\_status=False) | 直接调用 | 在get\_redis函数中，调用该函数，能够获取redis中的数据，并按照指定的格式进行转换，以及异常处理。  入参说明：  key：redis数据的key；  data\_type：数据类型，可选值为' json '、' str '、' int '、'float '、'bool '，'bytes'，该函数会将redis查询结果按data\_type进行转换，若为'bytes'则不会转换，因为redis查询结果本身就是bytes；  return\_with\_success\_status：函数返回值是否要有查询redis成功与否的状态标记。  返回值：  第一个返回值为redis查询结果（查询成功情况下），或者可以直接用于get\_redis函数返回的json（查询不成功情况下，有redis连接不成功、redis数据不存在、redis数据格式不正确等错误信息）；  如果return\_with\_success\_status为True，则会有第二个返回值，表面redis查询是否成功，一般需要对redis查询数据作进一步处理时，需要令return\_with\_success\_status为True，这样的话如果第二个返回值为False，可以直接return第一个返回值；如果第二个返回值为True，则可以进一步处理。 | 低 |
| BenchmarkApiView | 实例函数 | post\_model(self) | 重定义该函数或调用父类同名函数，benchmark框架在处理post请求时候调用该函数 | 处理post请求业务逻辑的函数。如果接口定义了primary\_model，而接口没有重定义该函数，则post请求会对primary\_model进行新增数据操作，接口请求字段和model中的字段一一对应。相对DRF的扩充：通过对请求体定义一个字典数组，来实现批量添加。 | 高 |
| BenchmarkApiView | 实例函数 | put\_model(self) | 重定义该函数或调用父类同名函数，benchmark框架在处理put请求时候调用该函数 | 处理put请求业务逻辑的函数。如果接口定义了primary\_model，而接口没有重定义该函数，则put请求会对primary\_model进行修改数据操作，接口请求字段和model中的字段一一对应。和DRF不同的是，简化了修改操作，DRF有两种修改操作的请求方式：put和patch，这两个在DRF的区别是它们在序列化检查字段是否存在时候的差别，put请求参数都为必填项，patch各个参数为可选项。而在实际应用中，DRF的patch方式会更多，并且能兼容DRF的put，而在大众理解层面，put更耳熟能详一些。所以benchmark使用put作为修改数据，其功能类似DRF的patch。 | 高 |
| BenchmarkApiView | 实例函数 | delete\_model(self) | 重定义该函数或调用父类同名函数，benchmark框架在处理delete请求时候调用该函数 | 处理delete请求业务逻辑的函数。如果接口定义了primary\_model，而接口没有重定义该函数，则delete请求会对primary\_model进行删除数据操作，接口请求字段和model中的字段一一对应。相对DRF的扩充：通过指定请求体中的pk数组（数组中的值为model的主键）支持批量删除。 | 高 |
| BenchmarkApiView | 静态函数 | get\_response\_by\_code  (code, msg, data, msg\_append) | 直接调用 | 在接口的get\_model、get\_redis、post\_model、put\_model、delete\_model函数中，在处理完接口逻辑后，接口若需要返回json，使用该函数，接口返回该函数的返回值。  入参说明：  code：响应状态码，默认为benchmark\_settings.py中定义的SUCCESS\_CODE；  msg：响应信息，默认为benchmark\_settings.py中定义的DICT\_RESPONSE\_BY\_CODE（框架内定义）和DICT\_CUSTOM\_RESPONSE\_BY\_CODE（平台自定义）中code对应的msg；  data：响应数据，默认为空；  msg\_append：默认为空，如果在使用默认msg后，在msg后面添加额外信息，可以使用这个参数。  返回值：  可用于接口返回的字典格式的数据，包括benchmark\_settings.py中定义的CODE、MSG、DATA字段（接口风格还未转换，下划线还未转驼峰）。 | 高 |
| BenchmarkApiView | 静态函数 | get\_http\_response\_by\_code  (code, msg, data, msg\_append) | 直接调用 | 该函数基本同get\_response\_by\_code，也是将入参封装成dict，不同的是再将dict封装json成Response类型，直接作为整个接口的返回向请求发起方发响应（这样会略过重命名、过滤字段等后续功能）。一般很少用此函数，只有需要对接口返回格式有定制的特殊情况才会使用。  入参说明：  同get\_response\_by\_code，略。  返回值：  将入参封装成Response类型，可以作为整个接口响应返回。 | 低 |
| BenchmarkModel | 类函数 | get\_model(cls, params=None, query\_set=None, select\_related=None, values=None, values\_white\_list=True, Qs=None, using='default', first=False, last=False, order\_by=None) | 直接调用 | 在benchmark框架内部的机制中，BenchmarkApiView的get\_model函数会调用primary\_model的get\_model函数获取数据。  而在实际的接口开发中，在接口的get\_model函数里，除了调用父类的get\_model，还能调用多继承了BenchmarkModel的model的get\_model函数，这样相对普通django model使用“.objects.filter()”能更便捷地获取到数据（已经将query set转换成字典，处理了各种异常，并且能直接作为接口数据返回）。  入参说明：  params：字典类型，同django model的“.objects.filter()”函数中的入参，支持各个model字段及其lookups字段的查询，也支持benchmark\_settings.py中定义的Q、Q\_OR、Q\_AND等查询函数；  query\_set：如果有需要在调用函数前就已经查询了一部分数据，则传入相应的query set，能够继续这个query set查询；  select\_related：类似django model操作的select\_related函数，是个select\_related多表关联查询的字段名字符串数组。除了支持普通django的select\_related只支持多对一（外键）、一对一、反向一对一查询，更支持一对多（反向外键）、多对多、反向多对多的多表关联查询；  values：返回字段过滤，功能同配置中的values，是个字段名的字符串数组；  values\_white\_list：返回字段过滤是否白名单过滤，功能同配置中的values\_white\_list，是否bool类型；  Qs：Q object查询，功能同配置中的Q；  using：model所在的数据库名，字符串类型；  first：是否取查询结果中的第一个数据，bool类型；  last：是否取查询结果中的最后一个数据，bool类型；  order\_by：类似django model的order\_by函数，按字段大学排序，字段名的字符串类型，如果是降序，字段名前加个“-”。  返回值：  可用于接口返回的字典格式的数据，包括benchmark\_settings.py中定义的CODE、MSG、DATA字段（接口风格还未转换，下划线还未转驼峰）。 | 中 |
| BenchmarkModel | 类函数 | post\_model(cls, data, user=None, using='default', serializer\_is\_custom=True) | 直接调用 | 机制基本同get\_model，但不是查询，而是对model的新增数据操作。  入参说明：  user：请求人，可以传入接口的self.user，用于写入创建者字段（需要配合MODEL\_CREATOR配置项一起使用）；  using：model所在的数据库名，字符串类型；  serializer\_is\_custom：则普通DRF序列化不会对带有主键的新增请求检查是否主键重复，如果该参数为True，表示插入前先检查是否有重复的主键，避免插入时候数据库报异常。  返回值：  可用于接口返回的字典格式的数据，包括benchmark\_settings.py中定义的CODE、MSG、DATA字段（接口风格还未转换，下划线还未转驼峰）。 | 低 |
| BenchmarkModel | 类函数 | put\_model(cls, data, user=None, using='default') | 直接调用 | 机制基本同get\_model，但不是查询，而是对model的修改数据操作。  入参说明：  data：字典类型，可以传入接口的self.data，并且需要确保该字典里面的key有需要修改的model字段的主键名（或者“pk”），对应的值为修改条目的主键值；  user：请求人，可以传入接口的self.user，用于写入修改者字段（需要配合MODEL\_MODIFIER配置项一起使用）；  using：model所在的数据库名，字符串类型。  返回值：  可用于接口返回的字典格式的数据，包括benchmark\_settings.py中定义的CODE、MSG、DATA字段（接口风格还未转换，下划线还未转驼峰）。 | 低 |
| BenchmarkModel | 类函数 | delete\_model(cls, data, user=None, using='default') | 直接调用 | 机制基本同get\_model，但不是查询，而是对model的删除数据操作。  入参说明：  data：字典类型，可以传入接口的self.data，并且需要确保该字典里面的key有需要删除的model字段的主键名（或者“pk”），对应的值为删除条目的主键值（删除单个），或多个主键值组成的数组（批量删除）；  user：请求人，可以传入接口的self.user，用于在使用标记位删除（MODEL\_DELETE\_FLAG=True）时候，写入删除者字段（需要配合MODEL\_MODIFIER配置项一起使用）；  using：model所在的数据库名，字符串类型。  返回值：  可用于接口返回的字典格式的数据，包括benchmark\_settings.py中定义的CODE、MSG、DATA字段（接口风格还未转换，下划线还未转驼峰）。 | 低 |
| BenchmarkModel | 静态函数 | model\_to\_dict(instance, fields=None, exclude=None) | 直接调用 | 基本同django.forms.models.model\_to\_dict，有两处不同的是：（1）如果配置了OMIT\_UN\_EDITABLE\_FIELDS=False，则会把所有只读字段也转换进dict返回（django的model\_to\_dict不会将只读字段返回），比如设置了时间字段为新建时自动写入，或者更新时自动写入；另外，对时间字段按类似“2017-10-01 08:00:00”这样的格式转化成字符串（日期和时间之间是个空格，使用DRF会是个“T”）。  入参说明：  instance：model的实例；  fields：字段名的字符串数组，白名单过滤；  exclude：字段名的字符串数组，黑名单过滤。  返回值：  转换为dict类型的model实例。 | 低 |

1. **benchmark接口实例变量一览**

在第2章中，有接口的类变量的介绍，这些类变量或实例变量是接口的配置项。而本章介绍的是接口的实例变量，这些变量和接口的请求参数相关。这些变量在接口处理的get\_model、get\_redis、post\_model、put\_model、delete\_model函数中可直接使用，如self.data即可取到请求体中的各个请求参数。

|  |  |  |
| --- | --- | --- |
| 变量名 | 值举例 | 变量含义 |
| request | 无 | 同DRF处理请求的函数中的request变量。 |
| method | ‘post’ | 同DRF的request.method，但为了开发上的方便，benchmark中各个变量、配置中的method都为小写。 |
| params | {‘employee\_id’: 1, ‘employee\_name’: ‘john’} | 请求url中（位于“?”后面）的参数，字典类型。若CONVERT\_KEYS=True，则已经转换成下划线格式。 |
| uri\_params | {‘id’: 1} | 请求uri中（位于“?”前面）的参数，字典类型。若CONVERT\_KEYS=True，则已经转换成下划线格式。 |
| data | {‘employee\_id’: 1, ‘employee\_name’: ‘john’} | 请求体中的参数，字典类型，支持json和form-data两种请求体。若CONVERT\_KEYS=True，则已经转换成下划线格式。 |
| user | 无 | 同DRF中的request.user。 |
| file | 无 | 上传的文件。 |
| redis | 无 | 接口定义了get\_redis函数的话，该变量即连接redis的实例，能进行各种操作。而在get\_redis函数中读数据的话，建议用 get\_redis\_value，该函数可以将redis的bytes数据转换成字符串、json解析等转换，并且有各种异常处理，得到的数据基本上能作为接口返回的数据。 |

1. **实际开发场景应用benchmark举例**

2、3、4章，已对benchmark目前所有功能相关的配置、函数、变量进行了介绍，多数功能能够参照2、3、4章的内容进行编码开发了。而本章结合接口开发中的实例，进一步详解一些表格中没法说尽的内容。

* 1. **BenchmarkApiView的父类View：PARENT\_VIEW**

BenchmarkApiView目前支持继承各种父类，有最早的APIView，后来增加了GenericAPIView、ViewSet、ModelViewSet，可以根据喜好，通过修改配置文件来修改父类，这样继承BenchmarkApiView的同时，也能使用其父类View的各种功能。目前测试发现ModelViewSet兼容swagger是最好的，所以如果需要使用swagger推荐使用ModelViewSet作为父类。在benchmark\_settings.py中配置的方法如下代码：

PARENT\_VIEW = 'ModelViewSet' # 可选值为 "APIView", "GenericAPIView", "ViewSet", "ModelViewSet"

* 1. **自适应model操作、自适应filter、自适应序列化：primary\_model**

primary\_model这个类变量，除了之前的一篇文章中的介绍，能无需任何代码，即可自适应地实现接口对model的增、删、改、查功能。另外，还能自适应地实现查询filter（如之前那篇文章介绍的那样，实际更能自适应支持lookups关键字查询，甚至多表关联查询）和序列化。而如果想自定义filter或序列化，则如普通DRF开发那样，定义类变量filter\_class或serializer\_class，即可覆盖benchmark的定义即可。

如5.5节的代码例子，只需定义primary\_model，而无需任何其他代码即可实现对primary\_model的查询filter、序列化。

* 1. **接口访问权限：access**

access是个接口的类变量，类型是个字典，字典的key有get、post、put、delete，分别对应查、增、改、删，字典的value可以是如下这些值，或以下值所组成的数组（支持多种身份的用户可访问接口的权限）：

None：该method不可用

'all'：该method可供任何人访问，包括没有登录过平台的请求方

'user'：该method可供任何登录过平台的请求方访问

'staff'：该method可供django user model中is\_staff字段为True的用户访问

'superuser'：该method可供django user model中is\_superuser字段为True的用户访问

'creator'：该method（get、put、delete）可供创建者查询、修改、删除，model中的创建者字段在MODEL\_CREATOR全局配置定义

如果自定义了USER\_RIGHT\_AUTHENTICATE\_FUNCTION，则可以使用该自定义权限认证函数中的身份标识的字符串。

如5.5节中的代码例子，接口的get请求需要用户登录，其他3种请求方式，需要平台管理员权限（该权限是平台自定义的权限，在相应的配置USER\_RIGHT\_AUTHENTICATE\_FUNCTION定义的自定义权限认证函数中有判断）。

* 1. **接口请求、响应参数名的驼峰、下划线转换：CONVERT\_KEYS、methods\_not\_need\_convert\_keys\_for\_request、methods\_not\_need\_convert\_keys\_for\_response**

如5. 5节中的代码例子，有配置PARENT\_VIEW = 'ModelViewSet'和CONVERT\_KEYS = True，无需额外代码（若使用常规的DRF开发方式，需要写序列化代码对其驼峰下划线转换、filter对model的各个字段进行查询等一些额外代码），即可实现对AppModule的增、删、改、查，并且兼容swagger。其中“Role.ADMIN”为用于使用USER\_RIGHT\_AUTHENTICATE\_FUNCTION中定义的平台自定义认证函数所允许的用户身份（比如自定义的User表有个字段为is\_admin的bool类型）。

有时候，部分接口请求或接口返回，不需要做转换（实际开发中有和bp、jira等其他平台交互的一些接口是这样），则可以定义benchmark view的类变量methods\_not\_need\_convert\_keys\_for\_request或methods\_not\_need\_convert\_keys\_for\_response，变量值为一个数组，数组中的字符串分别对应不需要转换的请求或响应的http method。比如下面这行代码是灰度平台响应bp的get请求接口，不能转驼峰，要用下划线：

methods\_not\_need\_convert\_keys\_for\_response = ['get']

* 1. **多表关联查询——select\_related、enable\_select\_related\_in\_params、enabled\_select\_related\_in\_params**

select\_related这个变量，除了之前的一篇文章中的介绍，在url中作为请求参数，而在实际开发中，更多的是定义在接口代码中，并且对接口调用方的请求更可控（通过接口请求参数中的select\_related可能查询到该请求方没有查询权限的model）。因此，最新版本的benchmark将select\_related进行了扩展，不但可以是请求url中的参数，也可以是类变量（对应该接口的所有请求），也能是实例变量（个别场景下，对应该接口内部业务逻辑，需要分不同情况分别处理每个请求不同的select\_related）。

另一方面，为了对接口的get请求查询权限的控制，使用enable\_select\_related\_in\_params和enabled\_select\_related\_in\_params对请求url中的select\_related参数值进行控制。如果需要启用（默认按全局配置不启用），则定义enable\_select\_related\_in\_params=True（启用），和enabled\_select\_related\_in\_params=['employee\_set']（启用时，可使用的字段），如果需要启用所有字段，则定义enabled\_select\_related\_in\_params='\_\_all\_\_'。

如下代码的select\_related定义，表示接口的get请求能查询到App表及其通过types、usergroup\_set和usergroup\_set\_\_members关联到的、总共4个model的数据，在自定义的get\_model函数中调用父类的同名函数，就能够获取到这4个model的数据（可用于接口返回的字典类型数据），然后对其结果经过一些业务逻辑代码的进一步过滤或处理，最终返回查询结果。

**class** AppView(BenchmarkAPIView):access = {  
 **'get'**: **'user'**,  
 **'post'**: (Role.ADMIN,), *# Role. ADMIN = 'admin' 平台自定义的管理员权限*  
 **'put'**: (Role.ADMIN,),  
 **'delete'**: (Role.ADMIN,)  
 }  
 primary\_model = App  
 select\_related = [**'types'**, **'usergroup\_set\_\_members'**]  
  
 **def** get\_model(self):  
 res = super().get\_model()  
 *# 省略一段业务逻辑的代码*  
 **return** res

* 1. **接口请求/响应参数重命名——rename\_params、rename\_uri\_params、rename\_data、rename\_fields**

在实际开发中，接口请求参数名、或响应字段，和model中的字段名需要不同，此时，可以使用rename\_params、rename\_uri\_params、rename\_data分别对请求url中的参数（url“?”后的）、请求uri中的参数（url“？”前的参数）、请求体中的参数，进行重命名。举例重命名params，如下定义类变量或实例变量rename\_params：

rename\_params={'id': 'department\_id', 'name': 'department\_name'} # 将参数名id变为department\_id，name变为department\_name。

这样，请求url中的id和name，在self.params中就成为department\_id和department\_name了。

rename\_uri\_params和rename\_data的使用方法，和rename\_params是类似的。

而如果要对接口返回结果中的字段进行重命名，则需要使用rename\_fields变量，该变量是对返回参数data中的字段，或data中的result（若data中有result）中的字段，进行重命名。另外，因为接口返回的有可能是嵌套数组或字典的有层次的json，则需要明确层次的路径，所以多了一个路径作为key，举例如下：

未使用rename\_fields时，接口返回如下（为了方便说明该变量的用法，假设不对接口返回参数名转换为驼峰，若要转换驼峰，rename\_fields的定义是完全一样的）：

{  
 **"code"**: 0,   
 **"msg"**: **"success"**,  
 **"data"**: {  
 **"employee\_id"**: 1,  
 **"employee\_name"**: **"EmployeeAX1"**,  
 **"employee\_info"**: {  
 **"age"**: 40,  
 **"sex"**: **"male"** },  
 **"department"**: 1,  
 **"projectteam\_set"**: [  
 {  
 **"project\_team\_id"**: 1,  
 **"project\_team\_name"**: **"ProjectTeam1"** },  
 {  
 **"project\_team\_id"**: 2,  
 **"project\_team\_name"**: **"ProjectTeam2"** }  
 ]  
 }  
}

定义rename\_fields变量如下，该变量也是个字典，rename\_fields字典的key为路径，根路径为“/”，遇数组或字典嵌套，则以数组或字典对应的key作为下一层路径（可以支持无限多层路径）；rename\_fields字典的value和rename\_params类似，是个命名转换的字典。如下定义意为将最外层的projectteam\_set重命名为project\_teams；将projectteam\_set中的project\_team\_id重命名为team\_id；将employee\_info中的age重命名为employee\_age：

rename\_fields = {  
 **'/'**: {**' projectteam\_set '**: **' project\_teams'**},  
 **'/projectteam\_set/'**: {**'project\_team\_id'**: **'team\_id'**},  
 **'/employee\_info/'**: {**'age'**: **'employee\_age'**},  
}

重命名后的接口返回如下：

{  
 **"code"**: 0,   
 **"msg"**: **"success"**,  
 **"data"**: {  
 **"employee\_id"**: 1,  
 **"employee\_name"**: **"EmployeeAX1"**,  
 **"employee\_info"**: {  
 **"employee\_age"**: 40, //重命名  
 **"sex"**: **"male"** },  
 **"department"**: 1,  
 **"project\_teams"**: [ //重命名  
 {  
 **"team\_id"**: 1, //重命名  
 **"project\_team\_name"**: **"ProjectTeam1"** },  
 {  
 **"team\_id"**: 2, //重命名  
 **"project\_team\_name"**: **"ProjectTeam2"** }  
 ]  
 }  
}

* 1. **接口响应参数过滤——values、values\_fields、values\_white\_list**

有时需要对接口响应参数进行过滤，去掉一些不需要返回的参数。有2种方式，使用values变量或values\_fields变量。它俩的区别是：

values：支持在get请求参数中的变量、接口的类变量、接口的实例变量，但只支持最外层字段的过滤（多表关联查询的一对多、多对多、反向多对多关系会令接口返回的数据层数增加）；

values\_fields：支持接口的类变量、接口的实例变量，支持类似rename\_fields那样多层的过滤。

而values\_white\_list是接口的类变量、接口的实例变量，bool类型，表示values或values\_fields变量是否白名单过滤（对请求url中的values，采用另一种方式，详见下面的例子），一般默认为True。

仍以5.6节中重命名前的接口返回为例，举例如下：

1. values过滤：

接口get请求url为：“……?values=[employee\_id,projectteam\_set]”（注意values的值不要有空格或引号），或在接口中定义变量values=['employee\_id', 'projectteam\_set']，并令values\_white\_list=True，表示只需要接口返回employee\_id和projectteam\_set两个最外层的字段及其下面的字段。

而如果使用黑名单过滤，并令其产生相同的效果，则接口get请求url为：“……?values=-[employee\_name,employee\_info,department]”（values的值最前面有个负号，并且数组中是黑名单的字段），或在接口中定义变量values=['employee\_ name', 'employee\_info', 'department ']，并令values\_white\_list=False，表示不需要接口返回employee\_ name、employee\_info和department三个最外层的字段及其下面的字段。

黑白名单两种方式的接口响应结果相同，如下：

{  
 **"code"**: 0,   
 **"msg"**: **"success"**,  
 **"data"**: {  
 **"employee\_id"**: 1,   
 **"projectteam\_set"**: [  
 {  
 **"project\_team\_id"**: 1,  
 **"project\_team\_name"**: **"ProjectTeam1"** },  
 {  
 **"project\_team\_id"**: 2,  
 **"project\_team\_name"**: **"ProjectTeam2"** }  
 ]  
 }  
}

1. values\_fields过滤：

如上例子使用values过滤，比较便捷，但没法支持内层字段（age、sex、project\_team\_id、project\_team\_name）的过滤，而使用values\_fields能够支持。values\_fields也是个字典，key为路径（和rename\_fields的key相同），值为路径下的过滤参数，和values的值相同，比如：

values\_fields = {  
 **'/'**: (**'employee\_name'**, **'projectteam\_set'**),   
 **'/projectteam\_set/'**: (**'project\_team\_name'**, ),  
}

values\_white\_list = **True**

以上是白名单过滤，黑名单过滤如下：

values\_fields = {  
 **'/'**: (**'employee\_id'**, **'employee\_info'**, **'department'**),  
 **'/projectteam\_set/'**: (**'project\_team\_id'**, ),  
}  
  
values\_white\_list = **False**

黑白名单接口返回相同，如下：

{  
 **"code"**: 0,   
 **"msg"**: **"success"**,  
 **"data"**: {  
 **"employee\_name"**: **"EmployeeAX1"**,  
 **"projectteam\_set"**: [  
 {  
 **"project\_team\_name"**: **"ProjectTeam1"** },  
 {  
 **"project\_team\_name"**: **"ProjectTeam2"** }  
 ]  
 }  
}

最后，需要说明的是：如果同时使用了rename\_fields和values或values\_fields的话，先进行字段过滤，再重命名。

* 1. **设置接口返回单个数据——get\_one**

如果uri中带有主键，比如下面这个url定义：

url(**r'^employee/(?P<pk>\d+)/?$'**, EmployeeView.as\_view(has\_pk=**True**)),

接口的uri\_params变量中会含有key值为“pk”的项，这样如果使用默认的benchmark框架的get\_model查询，接口返回的data直接就是那个查询结果的字典，比如下面这样：

{  
 **"code"**: 0,  
 **"msg"**: **"success"**,  
 **"data"**: {  
 **"employee\_id"**: 3,  
 **"employee\_name"**: **"EmployeeAY1"**,  
 **"employee\_info"**: null,  
 **"create\_time"**: **"2017-07-06 14:01:12.926816"**,  
 **"modify\_time"**: **"2017-07-06 14:01:12.926838"**,  
 **"department"**: 2  
 }}

而有些场景，get请求必填的请求参数，只有可能查询出最多一个数据，或者只需要第一个数据，并且使用的url不带pk，比如下面这个url定义：

url(**r'^employee/?$'**, EmployeeView.as\_view()),

接口查询结果的data是个带有数量和查询结果字段的字典，而结果字段是个数组，比如下面这样：

{  
 **"code"**: 0,  
 **"msg"**: **"success"**,  
 **"data"**: {  
 **"count"**: 1,  
 **"result"**: [  
 {  
 **"employee\_id"**: 3,  
 **"employee\_name"**: **"EmployeeAY1"**,  
 **"employee\_info"**: null,  
 **"create\_time"**: **"2017-07-06 14:01:12.926816"**,  
 **"department"**: 2,  
 **"modify\_time"**: **"2017-07-06 14:01:12.926838"** }  
 ]  
 }  
}

如果希望接口返回前一种形式（data直接就是那一条数据），则只需配置接口的类变量get\_one=True；或者也能视情况，设置其实例变量get\_one，返回两种形式的json格式之一，比如下面这段代码：

**class** EmployeeView(BenchmarkAPIView):  
 primary\_model = Employee  
  
 **def** get\_model(self):  
 **if 'employee\_id' in** self.params:  
 self.get\_one = **True  
 return** super().get\_model()

当请求参数中带有employee\_id时候，使用get\_one为True的格式，否则为默认的get\_one为False的格式。

* 1. **redis操作——get\_redis、get\_redis\_value**

若一个get请求查询的数据是在redis中，则可以重定义get\_redis函数来实现查询redis的操作。在get\_redis函数中，使用get\_redis\_value可以获取到相应的redis数据，redis可以存放benchmark直接返回的json，这样可以用return self.get\_redis\_value('redis\_data\_key')直接返回数据，这样如果查询redis有异常的话，也能返回相应的错误信息；如果查询到redis数据后需要对数据进一步处理，则可以令return\_with\_success\_status为True，并判断redis是否查询成功，如果不成功，则直接return错误的json。如下代码例子，先从redis中查询到average\_created\_count，并将其作为additional\_data：

**def** get\_redis(self):  
 redis\_res, success = self.get\_redis\_value(**'average\_created\_count'**, **'float'**, return\_with\_success\_status=**True**)  
 **if not** success:  
 **return** redis\_res  
 self.additional\_data = {**'average\_created\_count'**: redis\_res}  
 **return** self.get\_redis\_value(**'jira\_issue\_cc\_day'**)

1. **总结**

本文介绍的这些benchmark功能，是目前在实际开发中逐步积累而开发出的，今后会继续不断开发出各种方便易用的功能。我认为，任何软件开发中出现的重复的、相似性较大的代码段，都是需要尽可能地被复用的，benchmark也一直是以此作为宗旨，在实际开发中发现这些可以被复用的点，做进框架中去，从而不断提升开发效率和软件质量。