[Holmes] 基于 bodybuilder 通过对象传参构造 es DSL

修订历史

版本	修订日期	修订者	修订内容
1	2021.09.03	haoyang.huang@shopee.com	添加方案

背景

NodeJS 查询 es 依赖 bodybuilder，经由它构造查询 DSL。但具体使用有一些不便，例如无法过滤空值，以及某些用法使用不清晰，导致代码不连贯、可读性差。

因此在这基础上，通过传对象参数(满足自定义协议)做一层拦截处理，再通过 bodybuilder 构造 DSL。优化代码，增加可读性，也对 es 查询有更好的理解。

逻辑设计

DSL

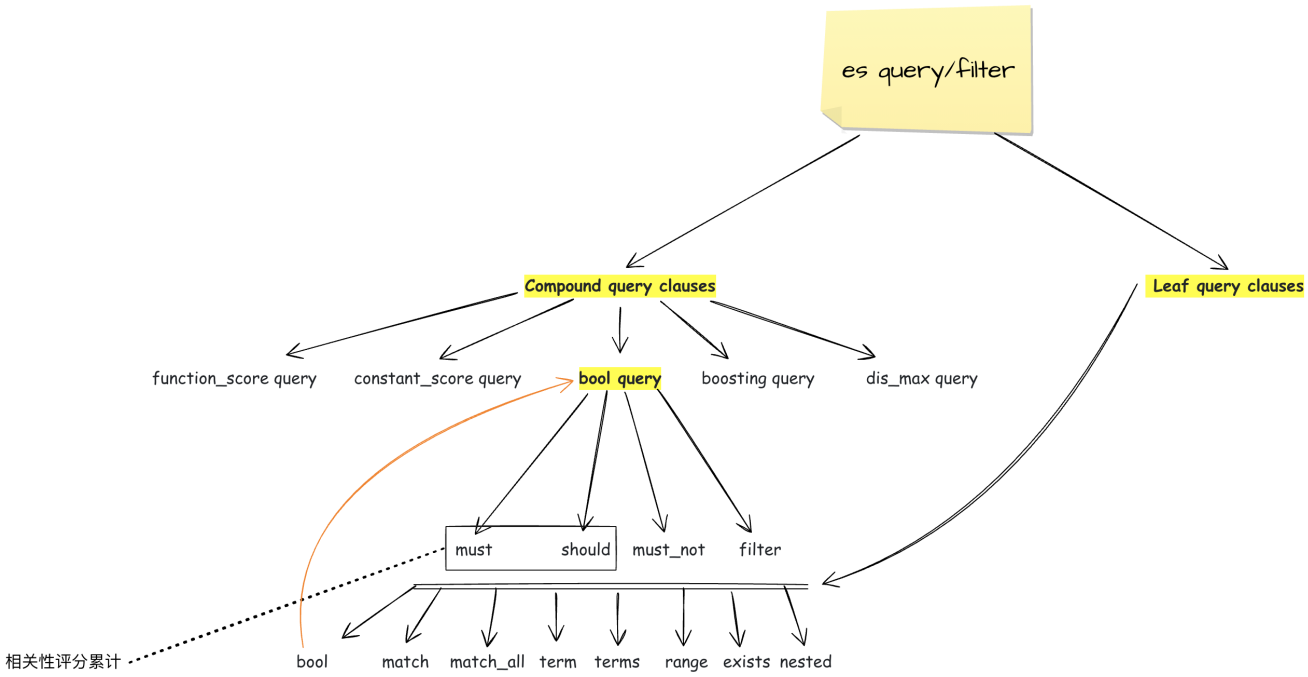
首先需要理解 es DSL(Domain Specific Language)，它是 es 提供的一个基于 JSON 查询语法，由两部分组成：**叶子查询子句(Leaf query clauses)**、**复合查询子句(Compound query clauses)**。

1. 叶子查询子句：查询条件的最小单位。
2. 复合查询子句：可以嵌套复合查询子句，直到最后由叶子查询子句构成。

es 查询主要有两种方式，**query** 和 **filter**。主要区别在 query 会根据计算 **相关性评分(_score)**，所以速度会更慢些。而且 filter 查询结果可以缓存，所以一般查询通过 filter 方式。除此之外，query 和 filter 的 DSL 基本一致。

前面说到 复合查询子句 可以嵌套，那时通过什么样的方式嵌套的？通过复合查询关键字：**bool**、boosting、constant_score、dis_max、function_score。

目前 Holmes 的查询较为简单，也跟 相关性评分 无关，复合查询只会用到 bool 关键字。

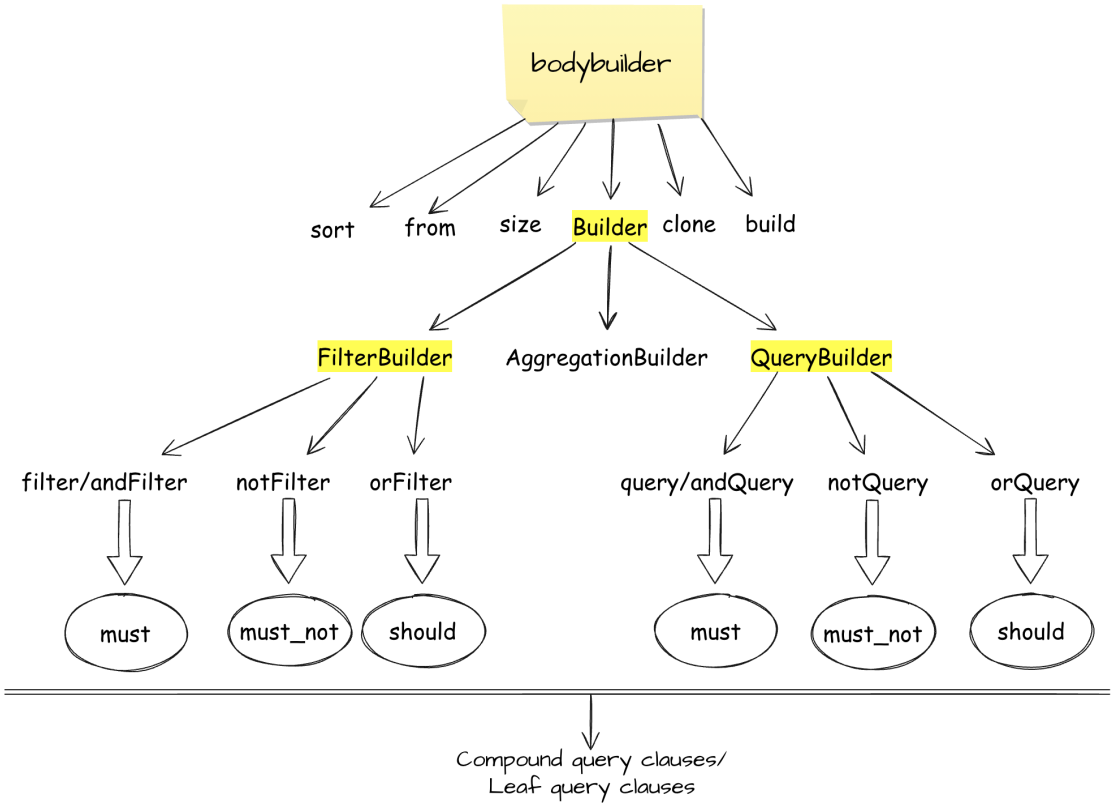


通过一个小例子说明上面规则：

```
{
    from: 0,
    size: 15,
    sort: [{ 'RiskReqHeader.ReqTime': { order: 'desc' } }],
    query: {
        bool: {
            filter: {
                bool: {
                    must: [
                        { term: { 'RiskReqHeader.AppID': 'spjup' } },
                        { terms: { CheckResult: [0] } },
                        { range: { 'RiskReqHeader.ReqTime': { gte: 1627747200, lte: 1628524800 } } },
                        {
                            bool: {
                                should: [
                                    {
                                        bool: {
                                            must: [
                                                { term: { 'RiskReqHeader.AppID': 'kredit' } },
                                                {
                                                    terms: {
                                                        'RiskReqHeader.SceneID': [10001,10003]
                                                    }
                                                }
                                            ]
                                        }
                                    }
                                ]
                            }
                        }
                    ],
                    must_not: [{ term: { 'RiskRspHeader.Result': 0 } }]
                }
            }
        }
    }
}
```

bodybuilder

然后我们再看，bodybuilder 怎么构造 DSL。



代码实现

TS 定义

```
import { FilterSubFilterFn } from 'bodybuilder'

export type TimeRange = {
  gt?: number
  gte?: number
  lt?: number
  lte?: number
  format?: string
}

type SortType = 'asc' | 'desc'

type SearchSort =
  | SortType
  | {
      order: SortType
      [k: string]: any
    }

export type SearchType =
  | 'match'
  | 'match_all'
  | 'term'
  | 'terms'
  | 'range'
  | 'exists'
  | 'nested'
  | 'bool'

export type FilterMatch = {
  [type in 'match' | 'match_all' | 'term']: { [field in string]?: string | number }
```

```
}

export type FilterTerms = {
  terms: {
    [field: string]: (string | number)[]
  }
}

export type FilterRange = {
  range: {
    [field: string]: TimeRange
  }
}

export type FilterExists = {
  exists: string[]
}

export type FilterBool = {
  bool: {
    fn: FilterSubFilterFn
  }[]
}

export type FilterNested = {
  nested: {
    path: string
    fn: FilterSubFilterFn
  }[]
}

export type FilterKey = 'and' | 'or' | 'not'

type SearchFilter = FilterMatch | FilterTerms | FilterRange | FilterExists | FilterBool

type SearchQuery =
  | FilterMatch
  | FilterTerms
  | FilterRange
  | FilterExists
  | FilterBool
  | FilterNested

export type SearchParams = {
  sort?: { [field: string]: SearchSort }
  from?: number
  size?: number
  filter?: { [key in FilterKey]?: Partial<SearchFilter> }
  query?: { [key in FilterKey]?: Partial<SearchQuery> }
}
```

buildDSL

```
checkValue(val: any) {
  return !['', undefined, null].includes(val)
}

buildDSL(params: SearchParams) {
  const { sort, from, size, filter, query } = params

  const dsl = EsBuilder()
  this.checkValue(from) && dsl.from(from!)
  this.checkValue(size) && dsl.size(size!)

  if (sort) {
    const sortArr = Object.entries(sort).map(([key, value]) => ({ [key]: value }))
    dsl.sort(sortArr)
  }
```

```
  Object.entries({ Filter: filter, Query: query }).forEach(([paramType, param]) => {
    param &&
      Object.entries(param).forEach(([key, searchFilter]) => {
        const filterFn = `${key}${paramType}` as 'andQuery' | 'notQuery' | 'orQuery'
        const { match, match_all, term } = searchFilter as FilterMatch
        const { terms } = searchFilter as FilterTerms
        const { range } = searchFilter as FilterRange
        const { exists } = searchFilter as FilterExists
        const { bool } = searchFilter as FilterBool
        const { nested } = searchFilter as FilterNested
        if (match) {
          Object.entries(match).forEach(([field, value]) => {
            this.checkValue(value) && dsl[filterFn]('match', field, value)
          })
        }
        if (match_all) {
          Object.entries(match_all).forEach(([field, value]) => {
            this.checkValue(value) && dsl[filterFn]('match_all', field, value)
          })
        }
        if (term) {
          Object.entries(term).forEach(([field, value]) => {
            this.checkValue(value) && dsl[filterFn]('term', field, value)
          })
        }
        if (terms) {
          Object.entries(terms).forEach(([field, value]) => {
            value.length > 0 && dsl[filterFn]('terms', field, value)
          })
        }
        if (range) {
          Object.entries(range).forEach(([field, value]) => {
            this.checkValue(value) && dsl[filterFn]('range', field, value)
          })
        }
        if (exists) {
          exists.forEach((field) => {
            this.checkValue(field) && dsl[filterFn]('exists', field)
          })
        }
        if (bool) {
          bool.forEach(({ fn }) => {
            this.checkValue(fn) && dsl[filterFn]('bool', fn)
          })
        }
        if (nested) {
          nested.forEach(({ path, fn }) => {
            this.checkValue(fn) && dsl[filterFn]('nested', 'path', path, fn)
          })
        }
      })
  })

  return dsl
}
```

兼容性

因为 bodybuilder 的链式调用特性，通过 **buildDSL** 方法返回的对象依旧可以使用 bodybuilder 的特性，无兼容问题。

遗留问题

基于目前业务较为简单，目前只处理了 **FilterBuilder** 和 **QueryBuilder** 的构造，然后支持的查询子句也有限。

如果需要拓展，需要开发人员根据业务需求去补充。