根据dify的#12092提交进行的代码分析

# 思路

• 从models开始看，看看数据库做了哪方面的更改

• 从controller开始看，看对外接口做了哪方面的更改（这里新增的接口都没列出来，感兴趣可自行研究）

• 从indexing模块开始看，看看在索引的时候有什么区别

• 从retrieval模块开始看，看看在召回的时候做了哪些操作

# models

新增了一张child\_chunks表

```python  
class ChildChunk(db.Model):  
 tablename = "child\_chunks"  
 table\_args = (  
 db.PrimaryKeyConstraint("id", name="child\_chunk\_pkey"),  
 db.Index("child\_chunk\_dataset\_id\_idx", "tenant\_id", "dataset\_id", "document\_id", "segment\_id", "index\_node\_id"),  
 )

```

那么他是怎么和父分段关联起来的呢，我们继续看

在分段下面多了一个属性，child\_chunks，由此可以猜测子分段是对segment进行的拆分

其实这里可以看到，FULL\_DOC模式与父子模式没关系

```python  
class DocumentSegment(db.Model): # type: ignore[name-defined]

```

rag中的models也有所改动，多了一个父子文档的绑定关系

```python  
class ChildDocument(BaseModel):  
 """Class for storing a piece of text and associated metadata."""

class Document(BaseModel):  
 children: Optional[list[ChildDocument]] = None

```

# controller

这里增加了个参数，点进去看即可知道，在更新segment的时候同时更新child模块

python  
class DatasetDocumentSegmentUpdateApi(Resource):  
 @setup\_required  
 @login\_required  
 @account\_initialization\_required  
 @cloud\_edition\_billing\_resource\_check("vector\_space")  
 def patch(self, dataset\_id, document\_id, segment\_id):  
 parser.add\_argument(  
 "regenerate\_child\_chunks", type=bool, required=False, nullable=True, default=False, location="json"  
 )  
 segment = SegmentService.update\_segment(SegmentUpdateArgs(\*\*args), segment, document, dataset)

```python  
 if document.doc\_form == IndexType.PARENT\_CHILD\_INDEX and args.regenerate\_child\_chunks:  
 # regenerate child chunks  
 # get embedding model instance  
 if dataset.indexing\_technique == "high\_quality":  
 # check embedding model setting  
 model\_manager = ModelManager()

```

# indexing

## index\_processor说明

上一个方法的输出为下一个方法的输入

extract方法提取文件内容------> transform格式化内容----->load 存储内容到向量数据库

## 内容理解说明

前面的探索发现知识更改了数据库方面的存储，看看index是做了什么吧

增加了一个processor的类，专门处理父子模式：core/rag/index\_processor/processor/parent\_child\_index\_processor.py

transform中，分别对待ParentMode.PARAGRAPH和ParentMode.FULL\_DOC，

他们的区别呢就是PARAGRAPH模式是对文档进行了切分，每一个分段都切分出相应的子模块。

而FULL\_DOC将全部文档作为一个上下文，所有的子块都属于一个父块

```python  
def transform(self, documents: list[Document], kwargs) -> list[Document]:  
 process\_rule = kwargs.get("process\_rule")  
 rules = Rule(process\_rule.get("rules"))  
 all\_documents = []  
 if rules.parent\_mode == ParentMode.PARAGRAPH:  
 # Split the text documents into nodes.  
 splitter = self.\_get\_splitter(  
 processing\_rule\_mode=process\_rule.get("mode"),  
 max\_tokens=rules.segmentation.max\_tokens,  
 chunk\_overlap=rules.segmentation.chunk\_overlap,  
 separator=rules.segmentation.separator,  
 embedding\_model\_instance=kwargs.get("embedding\_model\_instance"),  
 )  
 for document in documents:  
 # document clean  
 document\_text = CleanProcessor.clean(document.page\_content, process\_rule)  
 document.page\_content = document\_text  
 # parse document to nodes  
 document\_nodes = splitter.split\_documents([document])  
 split\_documents = []  
 for document\_node in document\_nodes:  
 if document\_node.page\_content.strip():  
 doc\_id = str(uuid.uuid4())  
 hash = helper.generate\_text\_hash(document\_node.page\_content)  
 document\_node.metadata["doc\_id"] = doc\_id  
 document\_node.metadata["doc\_hash"] = hash  
 # delete Splitter character  
 page\_content = document\_node.page\_content  
 if page\_content.startswith(".") or page\_content.startswith("。"):  
 page\_content = page\_content[1:].strip()  
 else:  
 page\_content = page\_content  
 if len(page\_content) > 0:  
 document\_node.page\_content = page\_content  
 # parse document to child nodes  
 child\_nodes = self.\_split\_child\_nodes(  
 document\_node, rules, process\_rule.get("mode"), kwargs.get("embedding\_model\_instance")  
 )  
 document\_node.children = child\_nodes  
 split\_documents.append(document\_node)  
 all\_documents.extend(split\_documents)  
 elif rules.parent\_mode == ParentMode.FULL\_DOC:  
 page\_content = "\n".join([document.page\_content for document in documents])  
 document = Document(page\_content=page\_content, metadata=documents[0].metadata)  
 # parse document to child nodes  
 child\_nodes = self.\_split\_child\_nodes(  
 document, rules, process\_rule.get("mode"), kwargs.get("embedding\_model\_instance")  
 )  
 document.children = child\_nodes  
 doc\_id = str(uuid.uuid4())  
 hash = helper.generate\_text\_hash(document.page\_content)  
 document.metadata["doc\_id"] = doc\_id  
 document.metadata["doc\_hash"] = hash  
 all\_documents.append(document)

```

可以看到多了一个方法\_split\_child\_nodes，他主要针对Document对象的pagecontent进行一个子分段拆分。

child\_splitter=child\_splitter = self.\_get\_splitter(.........)，主要使用这个对分段进行拆分，拆分最大长度为rules.subchunk\_segmentation.max\_tokens

但这里我们能看到，其实index\_processor并不会将父模块的内容存储到向量数据库中，向量数据库存储的都是child\_chunk的内容

```python

def \_split\_child\_nodes(  
 self,  
 document\_node: Document,  
 rules: Rule,  
 process\_rule\_mode: str,  
 embedding\_model\_instance: Optional[ModelInstance],  
) -> list[ChildDocument]:  
 child\_splitter = self.\_get\_splitter(  
 processing\_rule\_mode=process\_rule\_mode,  
 max\_tokens=rules.subchunk\_segmentation.max\_tokens,  
 chunk\_overlap=rules.subchunk\_segmentation.chunk\_overlap,  
 separator=rules.subchunk\_segmentation.separator,  
 embedding\_model\_instance=embedding\_model\_instance,  
 )  
 # parse document to child nodes  
 child\_nodes = []  
 child\_documents = child\_splitter.split\_documents([document\_node])  
 for child\_document\_node in child\_documents:  
 if child\_document\_node.page\_content.strip():  
 doc\_id = str(uuid.uuid4())  
 hash = helper.generate\_text\_hash(child\_document\_node.page\_content)  
 child\_document = ChildDocument(  
 page\_content=child\_document\_node.page\_content, metadata=document\_node.metadata  
 )  
 child\_document.metadata["doc\_id"] = doc\_id  
 child\_document.metadata["doc\_hash"] = hash  
 child\_page\_content = child\_document.page\_content  
 if child\_page\_content.startswith(".") or child\_page\_content.startswith("。"):  
 child\_page\_content = child\_page\_content[1:].strip()  
 if len(child\_page\_content) > 0:  
 child\_document.page\_content = child\_page\_content  
 child\_nodes.append(child\_document)  
 return child\_nodes  
```

# retrieval

讲完了存储，我们再来看父子模式是如何召回的吧

在：core/rag/retrieval/dataset\_retrieval.py中做了下结构调整，新增了一行代码

records = RetrievalService.format\_retrieval\_documents(newgrand\_documents)

注意，这行代码并不影响向量数据库的召回，向量数据库召回是通过single\_retrieve、multiple\_retrieve拿到结果的，records只是针对向量数据库召回结果进行了一个数据补充的处理

```python  
 def retrieve(  
 self,  
 app\_id: str,  
 user\_id: str,  
 tenant\_id: str,  
 model\_config: ModelConfigWithCredentialsEntity,  
 config: DatasetEntity,  
 query: str,  
 invoke\_from: InvokeFrom,  
 show\_retrieve\_source: bool,  
 hit\_callback: DatasetIndexToolCallbackHandler,  
 message\_id: str,  
 memory: Optional[TokenBufferMemory] = None,  
 ) -> Optional[str]:  
 .  
 .

```

这段函数的主要作用是：针对向量数据库返回的child chunk，找到其对应父块的segment信息，然后将子块的分数信息赋值给父块，这样的话，最终返回的结果就是：父块上下文+子块匹配的最高分数+匹配到的子块信息

```python  
@staticmethod  
def format\_retrieval\_documents(documents: list[Document]) -> list[RetrievalSegments]:  
 records = []  
 include\_segment\_ids = []  
 segment\_child\_map = {}  
 for document in documents:  
 document\_id = document.metadata["document\_id"]  
 dataset\_document = db.session.query(DatasetDocument).filter(DatasetDocument.id == document\_id).first()  
 if dataset\_document and dataset\_document.doc\_form == IndexType.PARENT\_CHILD\_INDEX:  
 child\_index\_node\_id = document.metadata["doc\_id"]  
 result = (  
 db.session.query(ChildChunk, DocumentSegment)  
 .join(DocumentSegment, ChildChunk.segment\_id == DocumentSegment.id)  
 .filter(  
 ChildChunk.index\_node\_id == child\_index\_node\_id,  
 DocumentSegment.dataset\_id == dataset\_document.dataset\_id,  
 DocumentSegment.enabled == True,  
 DocumentSegment.status == "completed",  
 )  
 .first()  
 )  
 if result:  
 child\_chunk, segment = result  
 if not segment:  
 continue  
 if segment.id not in include\_segment\_ids:  
 include\_segment\_ids.append(segment.id)  
 child\_chunk\_detail = {  
 "id": child\_chunk.id,  
 "content": child\_chunk.content,  
 "position": child\_chunk.position,  
 "score": document.metadata.get("score", 0.0),  
 }  
 map\_detail = {  
 "max\_score": document.metadata.get("score", 0.0),  
 "child\_chunks": [child\_chunk\_detail],  
 }  
 segment\_child\_map[segment.id] = map\_detail  
 record = {  
 "segment": segment,  
 }  
 records.append(record)  
 else:  
 child\_chunk\_detail = {  
 "id": child\_chunk.id,  
 "content": child\_chunk.content,  
 "position": child\_chunk.position,  
 "score": document.metadata.get("score", 0.0),  
 }  
 segment\_child\_map[segment.id]["child\_chunks"].append(child\_chunk\_detail)  
 segment\_child\_map[segment.id]["max\_score"] = max(  
 segment\_child\_map[segment.id]["max\_score"], document.metadata.get("score", 0.0)  
 )  
 else:  
 continue  
 else:  
 index\_node\_id = document.metadata["doc\_id"]

```

# 变更对比说明

## IndexingRunner.indexing\_estimate

### 1 documents获取方式

可以看到这里做了优化，之前使用的是一个通用的，需要适配多种情况，现在变为processor，不同的类型有不同的表现，符合面向对象编程的思想

### 2 对于普通分段模式处理

更改了返回值，返回值部分统一处理

### 3 对于qa分段处理

可以看到处理效果是一样的，只不过是对于qa文档的处理统一放在了transform里面

### 4 返回结构修改

注意：之前的preview\_textsl里面的内容是list[str]，现在改成list[Object]了

## DatasetRetrieval.retrieve

### segment获取方式修改

可以看到此次更改segment的获取方式改到RetrievalService.format\_retrieval\_documents 里面去了，封装成一个records对象返回

没有做任何更改

父子模式则是便利segments，为每个segments节点添加子节点信息

## VectorService.create\_segments\_vector

# 测试计划

```python  
def indexing\_estimate(  
 self,  
 tenant\_id: str,  
 extract\_settings: list[ExtractSetting],  
 tmp\_processing\_rule: dict,  
 doc\_form: Optional[str] = None,  
 doc\_language: str = "English",  
 dataset\_id: Optional[str] = None,  
 indexing\_technique: str = "economy",  
 ) -> IndexingEstimate:  
 # 正常模式已测试  
 # QA未测试

```

• 纯文本上传 默认规则 通过

• 纯文本上传 自定义规则 通过

• 文档带图片上传 默认规则 测试通过

• 文档带图片上传 自定义规则 测试通过

• 纯文本上传 qa模式自定义 不行，切换回原来的版本QA模式也用不了

• 文档带图片 QA模式默认 测试结果为空？