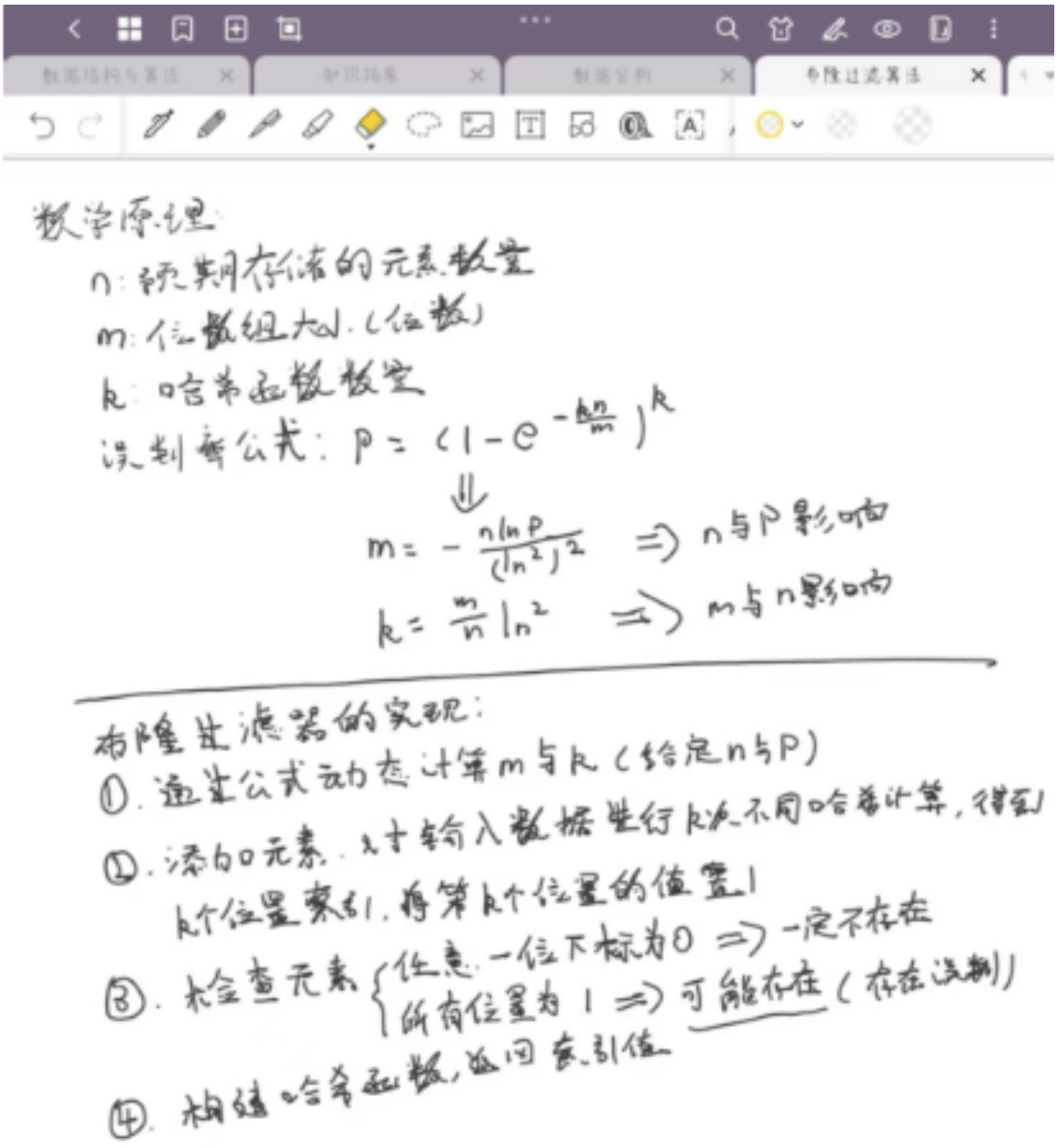
三种去重机制

1 三种去重机制的优缺点对比及应用场景

去重机制	优点	缺点	适用场景
集合去重	实现简单、查找准确、无误判	内存消耗大、不适合处理大规模数据	小规模数据去重、内存充足时、完全准确的去重需求
指纹去重	空间消耗小、速度快、适用于大数据处理	可能会有哈希碰撞（误判）、哈希算法质量影响准确性	文件去重、图片去重、大数据去重
布隆过滤器去重	空间效率极高、查询速度快、适合海量数据	假阳性（误判）、不支持删除、无法存储具体内容	海量数据去重、URL去重、缓存管理、爬虫去重

2 三种去重机制的原理

- 集合去重：
  - 将关键数据存放进集合中，通过集合内元素的唯一性实现去重
- 指纹去重：
  - 对集合去重的优化，通过计算数据的哈希值（即“指纹”）来唯一标识数据，将哈希值存到集合中，下一次判断集合中是否存在该指纹，如果不存在则再将哈希值存到集合中，指纹存在丢弃数据，指纹不存在则保留数据
- 布隆过滤器去重：



3 三种去重机制的代码实现

- set（集合）去重

```
# 准备数据
data = ['data1', 'data2', 'data1', 'data3']

# 初始化集合
seen = set()

# 将数据放入到集合中，通过集合内元素的唯一性实现去重
for i in data:
    seen.add(i)

# 打印去重结果
for i in seen:
    print(i)
```

运行结果

[2]:
```
# 准备数据
data = ['data1', 'data2', 'data1', 'data3']
```

[3]:
```
# 初始化集合
seen = set()
```

[4]:
```
# 将数据放入到集合中，通过集合内元素的唯一性实现去重
for i in data:
    seen.add(i)

# 打印去重结果
for i in seen:
    print(i)
```

```
data3
data2
data1
```

- 指纹去重

```
# 导包
from hashlib import md5

# 准备数据
data = ['data1', 'data2', 'data1', 'data3']

# 空容器，暂存数据
data_list = []

# 初始化集合
hash_seen = set()

# 将经过MD5处理后的数据放入到集合中，通过集合内元素的唯一性实现去重
for i in data:
    a = md5(i.encode()).hexdigest()
    if a not in hash_seen:
        hash_seen.add(a)
        data_list.append(i)

# 打印去重结果
for i in data_list:
    print(i)
```

运行结果

[34]:
```
# 导包
from hashlib import md5

# 准备数据
data = ['data1', 'data2', 'data1', 'data3']

# 空容器，暂存数据
data_list = []

# 初始化集合
hash_seen = set()

# 将经过MD5处理后的数据放入到集合中，通过集合内元素的唯一性实现去重
for i in data:
    a = md5(i.encode()).hexdigest()
    if a not in hash_seen:
        hash_seen.add(a)
        data_list.append(i)

# 打印去重结果
for i in data_list:
    print(i)
```

```
data1
data2
data3
```

- 布隆过滤器去重

```
import math
```

```python
import redis
import hashlib

class BloomFilter:
    # 初始化方法
    def __init__(self, n, p):
        # 初始化位数组长度 m
        self.m = self._calculate_m(n, p)
        # 预期存储的元素数量 n
        self.k = self._calculate_k(n, self.m)
        # 初始化存储位图的键
        self.redis_key = 'bloom_filter'
        # 初始化位数组
        self.redis = redis.Redis(host='localhost', port=6379, db=0)

    # 添加元素，对数据进行 k 次不同的哈希计算，并将第 k 位置1
    def add(self, item):
        for i in range(self.k):
            index = self._hash(item, i)
            self.redis.setbit(self.redis_key, index, 1)

    # 检查元素，判断元素是否一定不存在或者可能存在
    def contains(self, item):
        for i in range(self.k):
            index = self._hash(item, i)
            # 有一位为 0 则元素一定不存在，反之则可能存在
            if self.redis.getbit(self.redis_key, index) == 0:
                return False
        return True

    # 构造哈希函数，生成设置或检查索引
    def _hash(self, item, seed):
        # 实例化sha256对象
        h = hashlib.sha256()
        # 相当于对 item + seed 的组合计算哈希
        h.update(item.encode('utf-8'))
        h.update(str(seed).encode('utf-8'))
        # 返回 获取 SHA-256 的 32 字节哈希值，并将其高位在前转换成的大整数，对 m 取余将数据对应的每一位索引限制在位数组长度 m 中
        return int.from_bytes(h.digest(), byteorder='big') % self.m

    # 按照公式计算m（静态方法）
    @staticmethod
    def _calculate_m(n: int, p: float) -> int:
        m = -(n * math.log(p)) / (math.log(2) ** 2)
        return math.ceil(m)

    # 按照公式计算m
    @staticmethod
    def _calculate_k(n: int, m: int) -> int:
        k = (m / n) * math.log(2)
        return math.ceil(k)

if __name__ == '__main__':
    # 实例化布隆过滤器
    bf = BloomFilter(1000, 0.01)
```

```
    # 向空的布隆过滤器中添加元素
    for i in range(1000):
        bf.add('data' + str(i))
    #
    true_count = 0
    for i in range(1000):
        if bf.contains('not_data' + str(i)):
            true_count += 1

    print(f"误判率: {(true_count / 1000)*100:.2f}%")
```

运行结果

```
if __name__ == '__main__':
    # 实例化布隆过滤器
    bf = BloomFilter(1000, 0.01)
    # 向空的布隆过滤器中添加元素
    for i in range(1000):
        bf.add('data' + str(i))
    #
    true_count = 0
    for i in range(1000):
        if bf.contains('not_data' + str(i)):
            true_count += 1

    print(f"误判率: {(true_count / 1000)*100:.2f}%")
```

```
误判率: 1.00%
```

```
if __name__ == '__main__':
    # 实例化布隆过滤器
    bf = BloomFilter(1000, 0.05)
    # 向空的布隆过滤器中添加元素
    for i in range(1000):
        bf.add('data' + str(i))
    #
    true_count = 0
    for i in range(1000):
        if bf.contains('not_data' + str(i)):
            true_count += 1

    print(f"误判率: {(true_count / 1000)*100:.2f}%")
```

```
误判率: 5.20%
```