基于 Rust 语言的多维 Bloom Filter 设计与实现

引言

Bloom Filter 是由 Burton Howard BLoom 在 1970 年提出的一种用于数据去重，空间效率高的概率型数据结构。它专门用来检测集合中是否存在特定的元素。

Rust 语言是一门现代系统级编程语言，同时兼顾高性能和安全。和 C/C++ 相比，Rust 语言引入了所有权和生命周期机制，保证了系统运行时的安全性，与 Java/Go 相比，它没有 GC 机制，因此具有更高效的运行时系统。

这里笔者将基于 Rust 语言来实现一款 Bloom Filter 支持库，并拓展到多维，分析相关性能。

Bloom Filter 的设计与实现

设计思想

Bloom Filter（下文统一称为 BF）是由一个长度为 m 比特的位数组与 k 个哈希函数组成的数据结构。位数组均初始化为 0，所有哈希函数都可以分别把输入数据尽量均匀地散列。
当要插入一个元素时，将其数据分别输入 k 个哈希函数，产生 k 个哈希值，将 k 个哈希值对应的数据位置 1 。
当要查询一个元素的时候，同样将其数据输入哈希函数，然后检查对应的 k 个数据位，如果有任意一个数据位为 0，那么该元素一定不在集合中，否则该数据有较大可能性在集合中。

数据结构设计

对于一个 BF，我们完全可以在编译期就知道需要分配的内存空间是多少，因此我们可以使用 `常量泛型` 来实现这个数据结构。
2021 年 Rust 1.51 版本中稳定了常量泛型，它的一个作用是用于构建包含数组类型成员的结构体：

```
struct ArrayPair<T, const N: usize> {
    left: [T; N],
    right: [T; N]
}
```

从上面这个例子可以看到我们可以达到在编译期数组的长度是可变的，但是在运行时里面数组里面的数据是放到栈上的效果。
如果使用可变数组(Vec)去实现这种数据结构，那么数据将是放到堆上的，这样会损失一点运行时开销。
基于这种考虑我们使用常量泛型来实现 BF，结构体如下：

```
pub struct Filter<BHK: BuildHashKernels, const W: usize,
const M: usize, const D: u8> {
    buckets: Buckets<W, M, D>,      // filter data
    hash_kernels: BHK::HK, // hash kernels
    p: usize,              // number of buckets to decrement,
}
```

这个结构体定义除了三个常量泛型外还有一个 BHK 泛型，是和哈希函数有关的。
首先这个结构体由一个 Buckets 和一系列哈希函数组成。Buckets 里面包含数组结构，用于存放 BF 位数据信息，插入和查询都会从这里进行数据检索。
哈希函数将输入值散列，结果将会成为访问 Buckets 的下标。

Buckets 的设计与实现

BF 设计的核心是正是 Buckets 的设计，它的结构体实现如下：

```
pub struct Buckets<const WordCount: usize, const BucketCount:
usize, const BucketSize: u8> {
    data: [Word; WordCount],
    max: u8,
}
```

可以看到这个结构体有三个常量泛型，WordCount 表示数组里面的表项数目，BucketCount 表示 Buckets 中 Bucket 的数量，BucketSize 表示一个 Bucket 占多少字节。
数据检索的最小单位就是 Bucket，有时候我们需要大一点的 BucketSize。
通过常量泛型，我们可以在让 data 数组长度可变的同时，它的数据还是放在栈上。(如果使用 Vec 的话数据会放到堆上)
然后我们为这个结构体实现一系列的方法，向上提供一些 API：

```
impl<const WordCount: usize, const BucketCount: usize, const
BucketSize: u8> Buckets<WordCount, BucketCount, BucketSize> {
    /// 设置某个 bucket 的值
    pub fn set(&mut self, bucket: usize, byte: u8) {
        let offset = bucket * BucketSize as usize;
        let length = BucketSize as usize;
```

```
            let word = if byte > self.max as u8 { self.max } else
{ byte } as Word;
            self.set_word(offset, length, word);
        }

        /// 获取某个 bucket 的值
        pub fn get(&self, bucket: usize) -> u8 {
            self.get_word(bucket * BucketSize as usize,
BucketSize as usize) as u8
        }
}
```

通过这两个方法函数，我们就可以做到对某个 bucket 置 1 和获取某个 bucket 的值了。

哈希函数设计与实现

在 BF 中，我们需要的是一系列的哈希函数，而不是单个，因此我们借助 Rust 语言的迭代器语法来设计哈希函数：

```
/// A trait for creating hash iterator of item.
/// Rust 里面的 trait 相当于 Java 里面的 interface
pub trait HashKernels {
    type HI: Iterator<Item = usize>;

    fn hash_iter<T: Hash>(&self, item: &T) -> Self::HI;
}
```

这个 HashKernels trait 只有一个方法 hash_iter，它的语义是返回一个哈希函数的迭代器，这样就可以抽象出“一系列哈希函数”的概念了。

insert 方法和 contains 方法的实现

有了 Buckets 和 HashKernels 的基础，我们就可以实现数据的插入和查询方法，为应用场景提供 API 了。

```
impl<BHK: BuildHashKernels, const W: usize, const B: usize,
const S: u8> BloomFilter for Filter<BHK, W, B, S> {
    /// 插入数据，更新所有哈希结果对应的 bucket
fn insert<T: Hash>(&mut self, item: &T) {
        self.decrement();
        let max = self.buckets.max_value();
        self.hash_kernels.hash_iter(item).for_each(|i|
self.buckets.set(i, max))
    }
    /// 查询数据是否存在在集合中，只有所有哈希结果对应的 bucket
都被置一才返回 true
    /// 可能会误报，但不可能漏报
    fn contains<T: Hash>(&self, item: &T) -> bool {
        self.hash_kernels.hash_iter(item).all(|i|
self.buckets.get(i) > 0)
    }
}
```

这样我们就实现了 insert 和 contains 方法，可以插入和查询数据了。

正确性测试

这里基于 Rust 语言内置的单元测试系统，来测试上述结构实现的正确性：

```
fn _contains(items: &[usize]) {
        let mut filter = Filter::<_, {compute_word_num(730,
3)}, 730, 3>::new(0.03,
DefaultBuildHashKernels::new(random(), RandomState::new()));
        assert!(items.iter().all(|i| !filter.contains(i)));
        items.iter().for_each(|i| filter.insert(i));
        assert!(items.iter().all(|i| filter.contains(i)));
    }

    proptest! {
        #[test]
        fn contains(ref items in any_with::<Vec<usize>>
(size_range(7).lift())) {
            _contains(items)
        }
    }
```

测试结果：

```
running 1 test
test stable::tests::contains ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 6
filtered out; finished in 0.02s
```

多维 Bloom Filter 的设计与实现

在 Bloom Filter 基础上进行多维抽象

在 Rust 编程中，抽象是很重要的一个概念，好的抽象可以减少代码工作量。目前我们抽取多维 Bloom Filter 的 `多维` 概念，定义下面的 trait:

```
pub trait MultiBloomFilter {
    type BF: BloomFilter;
    type BI: IntoIterator<Item = Self::BF>;

    fn bloom_filter(self) -> Self::BI;
}
```

这个 trait 只有一个 bloom_filter 方法，这个方法会返回一个 `迭代器生成器`，迭代器迭代的是各个维度的 Bloom Filter 实例。通过这样的 trait 我们就很优雅地把多维 Bloom Filter 抽象出来了。

使用常量泛型实现多维 Bloom Filter

上面讲述过常量泛型非常适合用来实现编译期可以知道需要内存空间大小的数据结构，在多维 BF 的应用场景中，大部分的情况下数据的维度都是固定的，也就是说我们完全可以在编译期就知道多维 BF 中有多少个 BF。因此这里我们使用常量泛型来实现多维 BF：

```
pub struct DefaultMultiBloomFilter<BF: BloomFilter, const N:
usize> {
    bloom_filters: [BF; N]
}
```

上面结构体中的 BF 指的是具体 Bloom Filter 实现的类型，常量泛型 const N 指代的是维度。

为多维 Bloom Filter 实现迭代语法

这里为多维 BF 实现迭代器语法，可以使得代码编写更加方便：

```
impl<B: BloomFilter, const N: usize> IntoIterator for
DefaultMultiBloomFilter<B, N> {
    type Item = B;
    type IntoIter = std::array::IntoIter<Self::Item, N>;
    fn into_iter(self) -> Self::IntoIter {
        std::array::IntoIter::new(self.bloom_filters)
    }
}
```

上面的 into_tier 方法夺取多维 BF 的所有权，生成一个迭代器。

为多维 Bloom Filter 实现 MultiBloomFilter trait

上一节我们为多维 BF 实现了 IntoIterator trait，满足了实现 MultiBloomFilter trait 的条件，下面我们可以为多维 BF 实现该 trait，完成抽象和实现的统一：

```
impl<B: BloomFilter, const N: usize> MultiBloomFilter for
DefaultMultiBloomFilter<B, N> {
    type BF = B;
    type BI = std::array::IntoIter<B, N>;
    fn bloom_filter(self) -> Self::BI {
        self.into_iter()
    }
}
```

现在我们就可以通过 bloom_filter 方法来获取多维 BF 的迭代器了。

正确性测试

```
#[test]
fn default_multi_bloom_filter_test() {
    let filtes = [
        filter!(73, 3, 0.03,
DefaultBuildHashKernels::new(random(), RandomState::new())),
        filter!(73, 3, 0.03,
DefaultBuildHashKernels::new(random(), RandomState::new())),
        filter!(73, 3, 0.03,
DefaultBuildHashKernels::new(random(), RandomState::new()))
        ];
    let multi_filter = DefaultMultiBloomFilter::new(filtes);
    let items = [vec![1; 10], vec![1; 10], vec![1; 10]];
    let iter: Vec<_> = multi_filter
        .bloom_filter()
        .zip(items.iter())
        .map(|(mut f, i)| {
            i.iter().for_each(|item| f.insert(item));
            f
        })
        .collect();
    let ret = iter
        .iter()
        .zip(items.iter())
        .all(|(f, i)| {
            i.iter().all(|item| f.contains(item))
        });
    assert!(ret);
}
```

测试结果：

```
running 1 test
test multi::default_multi_bloom_filter_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 7
filtered out; finished in 0.00s
```

测试分析

这里借助开源项目criterion进行系统测试和分析。
该项目可以帮助我们运行测试任务，输出运行时间，统计尾延迟，运行时间最佳估计等。

延迟

基于下面几个可变参数对多维 BF 进行延迟测试：

- 数据的维度
- 数据的尺寸（数量）
- 可接受误报率

测试代码：

```
// 数据维度为 3，数据尺寸为 10，可接受误报率为 0.03
fn bench(c: &mut Criterion) {
    let multi = Fun::new("multi", |b, fp_rate| {
        let items0: Vec<usize> =
thread_rng().sample_iter(&Standard).take(7).collect();
        let items1: Vec<usize> =
thread_rng().sample_iter(&Standard).take(7).collect();
        let items2: Vec<usize> =
thread_rng().sample_iter(&Standard).take(7).collect();
        let items = [items0, items1, items2];
        b.iter(|| {
            let filtes = [
                filter!(73, 3, *fp_rate,
DefaultBuildHashKernels::new(random(), RandomState::new())),
                filter!(73, 3, *fp_rate,
DefaultBuildHashKernels::new(random(), RandomState::new())),
                filter!(73, 3, *fp_rate,
DefaultBuildHashKernels::new(random(), RandomState::new()))
            ];
            let multi_filter =
DefaultMultiBloomFilter::new(filtes);
            // 插入数据
```

```
            let iter: Vec<_> = multi_filter
                .into_iter()
                .zip(items.iter())
                .map(|(mut f, i)| {
                    i.iter().for_each(|item| f.insert(item));
                    f
                })
                .collect();
                // 查询数据
            let ret = iter
                .iter()
                .zip(items.iter())
                .all(|(f, i)| {
                    i.iter().all(|item| f.contains(item))
                });
            assert!(ret);
        });
    });
    c.bench_functions("multi", vec![multi], 0.03);
}
```

数据维度	数据尺寸	可接受误报率	最高延迟/us	最低延迟/us	延迟最佳估计/us
3	10	0.03	15.984	15.875	15.785
5	10	0.03	27.016	26.299	26.593
7	10	0.03	38.270	37.093	37.650
3	100	0.03	13.883	13.780	13.829
3	200	0.003	15.018	14.534	14.764
3	100	0.1	10.584	10.244	10.399
3	100	0.3	6.6112	6.3845	6.4917

分析：

- 数据维度增大，数据插入和检索的时间增大，延迟增大
- 数据尺寸增大，由于 Bloom Filter 的结构，访问数据的位置通过直接哈希得到，因此时间上变化不大
- 可接受误报率增大，延迟减少，但误报率也随之增大

false positive

false positive rate 指的是误报率，现在通过以下代码来测试 false positive rate:

```
fn bench(c: &mut Criterion) {
    let filtes = [
        filter!(730, 3, 0.03,
DefaultBuildHashKernels::new(random(), RandomState::new())),
        filter!(730, 3, 0.03,
DefaultBuildHashKernels::new(random(), RandomState::new())),
        filter!(730, 3, 0.03,
DefaultBuildHashKernels::new(random(), RandomState::new()))
    ];
    let multi_filter = DefaultMultiBloomFilter::new(filtes);
    let mut count = 0;
    let mut false_positives = 0;
    let mut iter: Vec<_> =
multi_filter.into_iter().collect();
    while count < 100000 {
        let items0: Vec<usize> =
thread_rng().sample_iter(&Standard).take(2).collect();
        let items1: Vec<usize> =
thread_rng().sample_iter(&Standard).take(2).collect();
        let items2: Vec<usize> =
thread_rng().sample_iter(&Standard).take(2).collect();
        let items = [items0, items1, items2];
        iter = iter
            .iter_mut()
            .zip(items.iter())
            .map(|(f, i)| {
                f.insert(&i[0]);
                f.clone()
            })
            .collect();

        iter = iter
            .iter_mut()
            .zip(items.iter())
            .map(|(f, i)| {
                if f.contains(&i[1]) { false_positives += 1;
}
```

```
                f.clone()
            })
            .collect();

        count += 1;
    }

    println!("MultiBloomFilter false positives: {:?}",
false_positives as f32 / 100000.0);
    let multi = Fun::new("multi", |b, _| b.iter(|| {}));
    let functions = vec![multi];
    c.bench_functions("multi false_positives_rate",
functions, ());
}
```

基本思想是先给多维 BF 插入一些数据，然后让它查询一些不存在的数据，统计误报的次数。
测试结果：

```
MultiBloomFilter false positives: 0.09344
```

空间开销

实际使用 Bloom Filter 时，一般会关注 false positive rate，因为这和额外开销相关。实际使用中，期望能给定一个 false positive rate 和将要插入的元素数量，能计算出分配多少的存储空间比较合适。
假设 BloomFilter 中元素总 bit 数量为 m,插入元素个数为 n，hash 函数个数为 k，false positive rate 记作 p，如果要最小化 false positive rate，可以有以下推导：

```
k = -lnp / ln2;
m = -n * lnp / (ln2) ^ 2
```

因此 Bloom Filter 需要的空间开销与预计插入元素个数，还有可接受误报率有关，关系如上式。
由于我在实现的过程中使用常量泛型来实现 Bloom Filter，因此需要的空间大小在编译期可以确定，数据可以放在栈上。

小结

本次实验我基于 Rust 语言实现了一款 Bloom Filter 的支持库，并且进行了多维的拓展。在编写代码的过程中使用了 Rust 语言刚稳定不久的常量泛型语法，尽量保证系统高性能的同时，确保内存安全。

该支持库有着良好的抽象，充分发挥了 Rust 语言的高性能，安全性和丰富的表现力，目前已在 github 上开源，欢迎访问：bloom-filters

通过本次实验，我对 Rust 语言的常量泛型语法更加熟悉了，对一些迭代器语法也写得更加熟练了，代码能力得到了不错的提高。希望后面继续精进。