|  |  |  |
| --- | --- | --- |
|  |  | 学号 1906010519 |
|  |  | 年级 2019级 |
| 说明: 河海大学  本科毕业论文 | | |
| 基于Rust的高并发场景技术探索与应用 | | |

|  |  |
| --- | --- |
| **专 业** | 计算机科学与技术 |
| **姓 名** | 李云汉 |
| **指导老师** | 陆佳民 |
| **评 阅 人** |  |

|  |
| --- |
| **2023年4月** |
| **中国 南京** |

|  |
| --- |
| BACHELOR'S DEGREE THESIS  OF HOHAI UNIVERSITY |
| Exploration and Implementation of High Concurrency Programming Based on Rust Programming Language |

|  |  |
| --- | --- |
| College : | College of Computer and Information |
| Major : | Computer Science |
| Name : | Li Yunhan |
| Directed by : | Lu Jiamin |

**NANJING CHINA**

郑 重 声 明

本人呈交的毕业论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计（论文）的研究成果不包含他人享有著作权的内容。对本设计（论文）所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计（论文）的知识产权归属于培养单位。

本人签名： 日期：

摘 要

Rust是一种多用途的编程语言，注重于安全与性能。首先，Rust拥有基于编译期静态检查的强大内存安全能力，保证所有指针（引用）都会指向有效的内存地址。其次，Rust使用“所有权”管理内存，使其在保证内存安全的同时可以拥有接近于C/C++的速度。同时，在并发编程方面，开发者号称借助其编译期执行的类型检查和所有权系统，Rust可以将很多并发错误在编译期间就被检查出来，做到“Fearless Concurrency”。

为了探索Rust在并发编程中的应用，进行实证研究，本文选择了一个并发算法——蓄水池抽样算法——并使用Rust和Java分别对其进行了实现。

本文所做的具体工作如下：

1. 介绍了并行蓄水池抽样算法，给出了一种使用阻塞队列管理线程的算法实现方案。
2. 使用Java实现了蓄水池算法，作为对照。
3. 使用Rust实现了蓄水池算法，探讨了实现过程中遇到的所有权、代码组织结构等问题，详细说明了Rust语言特性对于开发过程和结果的影响。
4. 对比了Rust和Java的开发过程和结果，并做简单的总结。

**关键词：**Rust编程语言、蓄水池抽样算法、并发程序设计

ABSTRACT

Rust is a multi-purpose programming language emphasizing on both safety and performance. Rust ensures memory safety with a set of strict security rules that are enforced at compile time. Thanks to ownership system, Rust allows programmers to write safe codes whose performance is close to C/C++ codes. As for parallel program design, developers of Rust claim that by leveraging ownership and type checking many parallel bugs are found during compile-time rather that runtime. This aspect of Rust is nicknamed “Fearless Concurrency”.

In order to conduct a preliminary empirical experiment on parallel programming with Rust, the following work has been done:

1. This paper introduces a way to implement parallel reservoir sampling algorithm, utilizing blocking queue for thread management.

2. Implementing reservoir sampling algorithm with Java for comparison.

3. This paper implements reservoir sampling algorithm with Rust and gives a detailed discussion on the language features utilized in the implementation.

4. Compares coding experience and program performance of Rust and Java. Lastly, a summary is given.

**Key words:** Rust programming language; reservoir sampling algorithm; parallel programming

目录

[摘 要 I](#_Toc134175885)

[ABSTRACT II](#_Toc134175886)

[目录 III](#_Toc134175887)

[第1章 绪论 1](#_Toc134175888)

[1.1 研究背景 1](#_Toc134175889)

[1.2 研究现状 1](#_Toc134175890)

[1.3 研究成果与意义 2](#_Toc134175891)

[第2章 Rust及其语言特性概述 3](#_Toc134175892)

[2.1 Rust的历史 3](#_Toc134175893)

[2.2 Rust的语言特性 3](#_Toc134175894)

[2.2.1 内存安全：所有权与生命周期 3](#_Toc134175895)

[2.2.2 抽象特性：Rust的面向对象特性与函数式编程 4](#_Toc134175896)

[2.2.3 并发特性：基于OS线程的并发编程、异步编程与可扩展并发编程 6](#_Toc134175897)

[2.3 Rust的性能 7](#_Toc134175898)

[第3章 蓄水池算法 9](#_Toc134175899)

[3.1 朴素蓄水池算法 9](#_Toc134175900)

[3.1.1 算法过程 9](#_Toc134175901)

[3.1.2 算法证明 11](#_Toc134175902)

[3.2 并行蓄水池算法 12](#_Toc134175903)

[3.2.1 合并算法 12](#_Toc134175904)

[3.2.2 合并算法证明 14](#_Toc134175905)

[3.2.3 协调线程之间的合作 14](#_Toc134175906)

[第4章 基于Java的蓄水池算法实现 17](#_Toc134175907)

[4.1 数据结构 17](#_Toc134175908)

[4.2 算法实现 17](#_Toc134175909)

[4.2.1 传递取样结果 17](#_Toc134175910)

[4.2.2 协调合并线程 18](#_Toc134175911)

[4.3 代码组织 21](#_Toc134175912)

[4.3.1 SimpleReservoir<T>类 21](#_Toc134175913)

[4.3.2 SampleResult<T>类 22](#_Toc134175914)

[4.3.3 SamplerHandle<T>类 22](#_Toc134175915)

[4.3.4 ParallelReservoir<T>类 22](#_Toc134175916)

[第5章 基于Rust的蓄水池算法实现 25](#_Toc134175917)

[5.1 数据结构 25](#_Toc134175918)

[5.2 算法实现 25](#_Toc134175919)

[5.2.1 传递取样结果 25](#_Toc134175920)

[5.2.2 协调合并线程 26](#_Toc134175921)

[5.3 Rust的编程范式 27](#_Toc134175922)

[5.3.1 并发编程范式 27](#_Toc134175923)

[5.3.2 代码复用 28](#_Toc134175924)

[5.3.3 类型系统与特性约束 31](#_Toc134175925)

[5.3.4 Rust的错误传递 33](#_Toc134175926)

[5.4 Rust与Java代码的性能对比 34](#_Toc134175927)

[5.4.1 测试平台与方法 34](#_Toc134175928)

[5.4.2 性能测试结果 35](#_Toc134175929)

[5.4.3 使用特征对象带来的性能波动 35](#_Toc134175930)

[5.5 代码的正确性验证 38](#_Toc134175931)

[5.6 本章总结 39](#_Toc134175932)

[第6章 总结 41](#_Toc134175933)

[6.1 主要成果 41](#_Toc134175934)

[6.2 未来展望 41](#_Toc134175935)

[参考文献 42](#_Toc134175936)

# 绪论

## 研究背景

Rust作为一门新兴的高级语言，以其接近C/C++运行效率、基于强大的编译器静态检查能力的内存管理等特点获得了业内广泛的关注，并同时取得了来自学院派和工业界的认可：Linux kernel在6.1版本引入了对于Rust编程语言的支持；同时包括Microsoft在内的各大厂商也在着手对Rust进行技术储备。

作为一门系统级语言，高效的并发处理是Rust的主要目标之一。Rust原生支持OS线程级的并发编程，并且支持async/await并发模型。同时，Rust的开发者宣称，通过高效地利用Rust编译器的所有权系统与类型检查，Rust可以将许多运行时出现的错误转化为编译期就可以发现的错误，从而提高并行程序的开发效率。 (1)

## 研究现状

首先，Rust是一门新兴语言，学习Rust时的主要知识来源是GitHub在线开源官方文档。如Rust语言圣经（中文） (2)、The Rust Programming Language（英文） (1)、Asynchronous Programming in Rust（英文） (3)等。

Rust的优秀表现引起了一些研究者的兴趣。对于Rust的论文主要可以分为以下三类：

1. 关于Rust语言正确性的形式化验证。
2. 经验主义/实证研究。（占多数）
3. 应用Rust进行实际开发

Boqin Qin等 (4)、秦伯钦等 (5)以及Zemin Yu等 (6)则对于Rust在并行编程、unsafe代码段等实践中的安全问题进行了基于代码分析的经验主义/实证研究。其中秦伯钦等还基于研究成果提出了一种Rust非阻塞性Bug的检测办法；Zemin Yu等发现Rust并不像开发者宣称的那样支持“Fearless Concurrency”开发：Rust编译器不能保证消除所有数据竞争错误（data racing）。牛聚川等提出并实现了一种融合模糊测试和形式化验证的Rust测试工具 (7)。王烁程等提出了基于Rust Async/Await的基于协程的飞地内部调度方案，并设计了调度机制 (8)。

## 研究成果与意义

首先，在第2章中笔者对Rust的基本特性进行了介绍。

为了从语言特性、抽象层次、并发性能等层面对比新兴的Rust语言和传统的Java语言，本论文中选择了“蓄水池取样算法”这一并发算法，并分别用Java语言的线程库（第4章）和Rust语言OS线程级的并发模型（第5章）对其进行了实现。并据此比较两个语言在开发过程和开发结果上的差异。

尽管本文所作研究较为简单，但记录了代码编写中遇到的一些问题，这些问题中很多都与Rust的类型检查、所有权等语言特性相关。这为以后编写Rust程序积累了经验。

同时，本文发现了使用特征对象带来的性能波动，但在解释这种性能波动时遇到了问题（5.4.3）。

# Rust及其语言特性概述

## Rust的历史

Rust是一种由Graydon Hoare设计，Mozilla后由接手的的高级编程语言。由Mozilla于2010年正式发布，于2015年发布了第一个稳定版（Rust 1.0 stable release），并在2016年及以后趋于稳定。Rust的发展经历了多次重要迭代，包括引入了特性（trait）、类（class）、析构函数（destructor）等语言特性，以及Non-Lexical Lifetime优化（即NLL优化，这种优化专门用于找到某个引用在作用域结束前就不再被使用的代码位置）等编译器优化。

疫情期间，Mozilla出于成本控制等考虑，解散了Servo（一个基于Rust的浏览器引擎）的开发小组。由于小组成员中很多都是Rust社区的核心成员，这对于Rust的发展产生了很大冲击。但令人欣慰的是，2021年Rust基金会由AWS、Huawei、Google、Microsoft、和 Mozilla等企业宣布注资成立。此基金会负责管理Rust的商标，并承担运营社区的财务职责。

## Rust的语言特性

尽管Rust与C/C++、Java具有很强的相似性，但其优秀的安全性和抽象能力来自于Rust的一些设计良好的语言特性。

### 内存安全：所有权与生命周期

Rust非常擅长保证内存安全：这种语言依赖强大的编译器静态检查来保证对于所有的引用都指向内存中的合法资源。这种由编译器负责的静态检查不像Java、Go等语言的运行时垃圾回收（GC，Garbage Collection）——不需要在程序运行时消耗资源，因此具有较好的性能；同时也不需要像C/C++一样手动管理内存，因此具有较高的安全性。

Rust使用所有权的概念来管理内存。简单来说，内存（堆空间）中的每一个资源（值，value）都为一个变量（variable）所拥有，这个变量称为该内存资源的所有者（owner of the value）。一个内存资源只能有一个所有者。当所有者离开作用域时，其所拥有的内存资源被释放（drop）。

可以说，Rust通过编译器静态检查的方式，强制程序员使用了RAII （Resource Acquisition Is Initialization，资源获取就是初始化）的概念。

除了所有权，Rust还有生命周期（Lifetime）的概念。想要使用某个内存资源，除了获得其所有权，还可以对其进行借用，获得指向这个资源的一个不可变或可变的引用。Rust编译器会在编译期间检查对内存资源的引用是否会超出拥有这个资源的变量的生命周期，以避免在使用引用时，其对应的资源被错误地提前释放了。

通过所有权与生命周期，Rust编译器保证了内存安全。

### 抽象特性：Rust的面向对象特性与函数式编程

尽管Rust支持一些面向对象特性：比如支持在结构体（struct）中封装一些外部不可见的成员变量和成员函数（封装），但Rust并没有像C/C++、Java那样支持类（class）之间的继承。Rust通过为类添加特征（trait）的方式支持面向对象编程中的继承和多态。

Rust的设计者似乎不喜欢继承关系（不喜欢is-a关系），并且认为没有继承的Rust通过特征（Trait）和结构体（struct）也可以实现多态等面向对象特性。（更倾向于has-a关系）。

Rust的开发者时这样解释的： (1)

继承一般有两个作用。第一个是复用父类的代码。Rust中，特征的方法可以有默认实现，这可以为程序员提供有限的代码复用功能。第二个作用是多态，即同一方法对于不同的类型有不同的行为。在Rust中，多态可以有效地通过泛型（Generics）加特征约束（Trait Bounds）实现。Rust可以为实现了特定特征的类型实现特定的行为；可以为实现了不同特征的类型提供不同行为。（具体可见2节）

特征类似于Java中的接口（interface），特征定义了一组可以被共享的行为，只要实现了特征，你就能使用这组行为。换句话说，拥有某个特征的类需要实现这个特征所定义的方法。特征中也可以定义具有默认实现的方法。通过特征（trait），Rust实现了代码的复用，间接地支持了继承。尽管这种方式是比较啰嗦的，因为特征虽然支持多态（默认实现的方法中可以调用没有默认实现的方法），但接口默认实现的方法使用起来没有继承方便，需要手动为每个拥有特征的类实现特征所定义的方法。

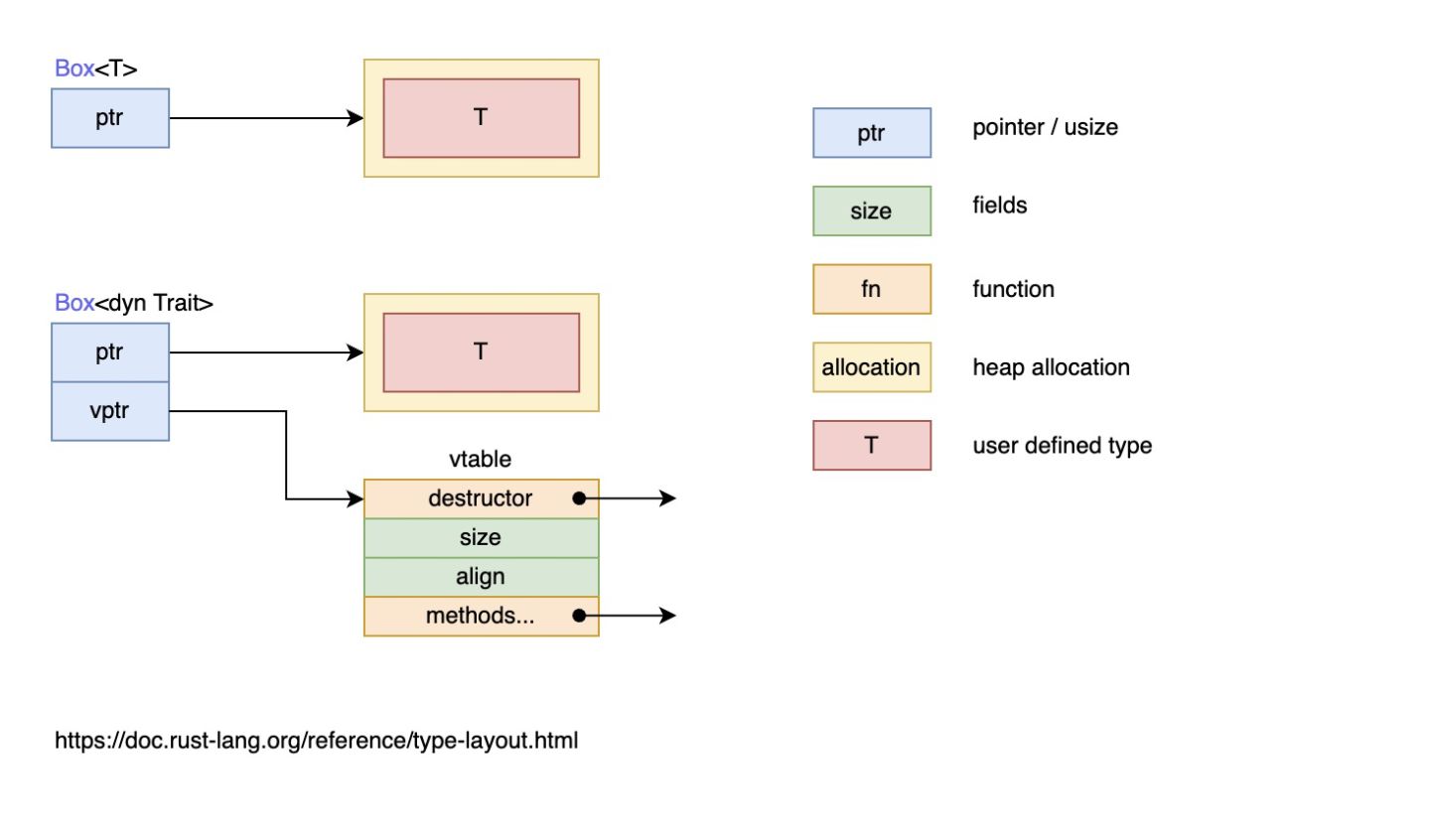
需要指出的是，一些程序员使用Rust的Deref特征来间接地模拟了继承功能。但一些人认为其为一种反模式（背离Rust设计初衷的做法），是一种误用编译器特性的错误做法 (9)。其问题包括但不限于：只支持单继承、不能继承父类的特征等。

此外，特征有标记的功能（marker trait）。举例来说，Rust多线程编程中，编译器仅允许在线程之间传递拥有Sync特征的类型的引用；仅允许在线程之间传递拥有Send特征的类型的所有权。但这两个特征并没有定义任何的方法。

Rust类型系统还支持特征对象（Trait Object），这是一种动态分发（dynamic dispatch）方式的多态。程序员不需要知道一个特征对象的类型究竟是什么，而只需要知道这个类型具有哪些特征，具有哪些行为。因此对于一个特征对象，它支持且仅支持其对应的特征所定义的行为。

举例来说，函数参数可以是一个指向了实现了某种特征的类型（的指针，Box<dyn Trait>），以使该函数对于不同的类型（但这些类型有相同的特征）可以有不同的行为。

在5.4.3节中描述了在测试过程中发现的一个由使用特征对象导致的性能波动。



图片 2‑1 由智能指针封装的特征对象Box<dyn Trait>

除此以外，Rust还支持函数式编程。在写Rust代码的时候，程序员可以使用闭包（closure，与Java中的lambda表达式类似）、模式匹配、迭代器等语言特性来简化自己的代码。

举例来说，在创建线程时，我们可以使用闭包描述线程的行为，并方便地使用闭包的特性向线程中传递变量的所有权。

### 并发特性：基于OS线程的并发编程、异步编程与可扩展并发编程

Rust的开发者宣称，通过高效地利用Rust编译器的所有权系统与类型检查，Rust可以将许多（其它语言中）运行时出现的错误转化为编译期就可以发现的错误，从而提高并行程序的开发效率。

举例来说，如果程序员试图在线程之间传递某个变量的引用，而又不能保证这个变量的生命周期覆盖了两个线程（这个变量可能在引用失效前被释放），那么Rust编译器就会出手阻止，给出“borrowed value does not live long enough”的错误。

尽管如此，Rust的大部分并发特性不是来源于Rust语言本身，而是来自Rust的标准库和第三方库。Rust标准库支持基于OS线程的并发编程，程序员可以创建操作系统级的线程来完成他们的任务。Rust鼓励程序员使用通道传递消息，并在标准库中提供了通道std::sync::mpsc，其中mpsc是multiple producer, single consumer的缩写，代表了该通道支持多个发送者，但是只支持唯一的接收者。

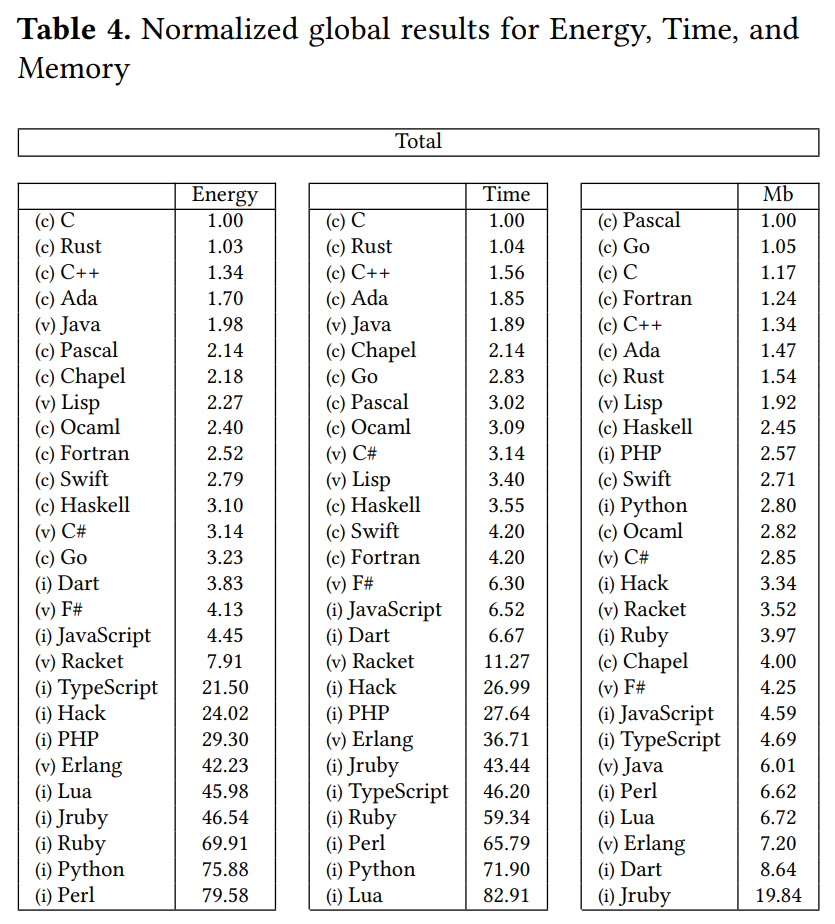
通道有两种：同步通道和异步通道。对于同步通道，发送者发送消息后会被阻塞，只有在消息被接收后才解除阻塞；而异步通道不会阻塞发送者。 (2)

同时，Rust通过语言特性（Trait等）、标准库和第三方库（tokio等）支持async/await并发模型。

此外，Rust语言本身支持可扩展的并发编程。在std::marker中有两个标记特征（marker trait）：Send和Sync。Rust编译器的类型检查和特征系统仅允许在线程之间传递拥有Send特征的类型的所有权；仅允许在多个线程中通过引用访问拥有Sync特征的类型。Rust的基础类型一般都拥有Send和Sync特征，而手动为某种类型标记Send和Sync特征是不安全的（Unsafe），因为这会绕过编译器的检查。 (1) 在这两个标记特征的基础上，我们可以扩展标准库和第三方库所提供的并发功能。

## Rust的性能

据Rust的开发者称，Rust具有媲美C/C++的性能。一些研究也已经证明，Rust的性能十分出色。在综合了27种编程语言针对10个编程问题的性能表现后，Pereira, Rui等总结出了以下表格。 (10)



图片 2‑2 归一化后的各语言能量、时间、内存使用量对比

从表格中不难看出，Rust在能耗、速度上占据很大优势，甚至拥有比C++更接近于C的性能。

# 蓄水池算法

本章介绍了朴素蓄水池算法的算法过程与证明流程，并进一步引入了基于朴素蓄水池算法的并行蓄水池算法，给出了其流程和证明。

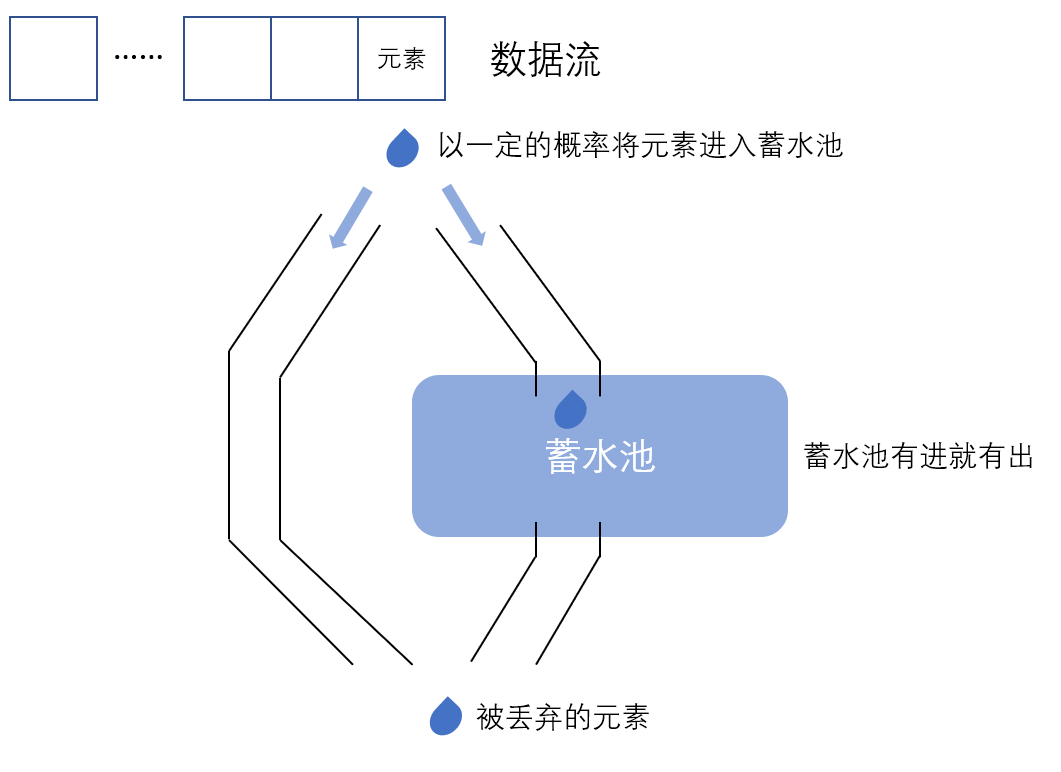
## 朴素蓄水池算法

蓄水池算法是一种抽样算法，它可以用于在数据流中抽取个元素，并确保数据流中的每一个元素可以以相同的概率被抽取到（其中数据流指只能以实现定好的顺序依次读取的元素序列，假设共有个元素，是一个很大的数字）。其时间复杂度为，空间复杂度为。

蓄水池算法的应用场景很多。比如说，现需要对直播间弹幕进行分析，则可以使用蓄水池算法从中抽取出一定数量的弹幕进行分析。

### 算法过程

简单说，朴素蓄水池算法维护一个长度为k的数组（vector），算法名字中的“蓄水池”就代指我们维护的这个数组。对于每一个数据流中的元素，以一定的概率决定其是否被取样。倘若决定取样，则用这个元素添加进数组中。数组（蓄水池）的大小是不变的，因此有进就有出：其中的一个旧元素被替换为新元素，旧元素被丢弃。（如图片 3‑1描述蓄水池算法的简图所示）



图片 3‑1描述蓄水池算法的简图

假设数据流的规模为N，需要取样的数量为k，蓄水池算法的流程如下：

1. 首先创建一个可以容纳k个元素的数组
2. 对于序列中的第个元素，全部放入数组中
3. 从序列中的第个元素开始，以的概率来决定该元素是否被替换到数组中（数组中的元素被替换的概率是相同的，即每个元素被替换的概率均为）。
4. 遍历完所有元素后，数组中剩下的元素即为算法结果，即取样结果。

伪代码如下所示（Algorithm 3‑1 朴素蓄水池算法）：

Algorithm 3‑1 朴素蓄水池算法

|  |  |
| --- | --- |
| Algorithm 朴素蓄水池算法 | |
|  | Array[k] samples; |
|  | function try\_sample(item){ |
|  | if (decided not to sample the item){ |
|  | return false |
|  | }else{ |
|  | index = randomly generate an unsigned integer in {1, 2, …, k} |
|  | samples[index] = item; |
|  | return true; |
|  | } |
| 10. | } |

其中，当try\_sample()返回true时，表示new\_element被取样，但不保证新的元素来之后仍被取样；当返回false时表示new\_element没有被取样。

### 算法证明

**PART1**

对于数据流的前个元素，即元素，其被囊括进数组的概率为1。数组被前个元素填满。

决定第个元素是否进入数组时，中的被取代的概率为选中的概率乘以选择作为被替代元素的概率，即：

则在算法试图取样时候被保留的概率为：

最终，处理完到第个元素后，被保留的概率为

**PART2**

对于数据流中中的某个元素，他们一开始不在数组内。被取样的概率为。倘若被囊括进了数组，则当试图取样第个元素时，同理保留的概率为

所以最终留在数组内的概率为：

以上过程证明，数据流中每个元素被取样的概率相等，均为。

## 并行蓄水池算法

现在我们考虑这样一种分布式/并行的情况：一个很大的数据流需要被分为多个数据流，由多个线程/分布式服务器分别取样，再将结果汇总，并且每个子数据流的元素个数可能不等（但都远大于k）。

并行蓄水池算法的重点在于合并取样结果。我们定义**取样结果**包含被取样的元素组成的数组以及数据流中元素的总数。合并算法指：将来自两个数据流的取样结果合并为一个，使得合并后的取样结果等价于将数据流合并为，并使用朴素蓄水池算法进行取样后得到的取样结果。

如果能合并两个数据流的取样结果，我们就可以将朴素蓄水池算法的取样结果两两合并，借此将其推广到多个数据流的并行蓄水池算法。

### 合并算法

以下主要介绍合并算法的流程。

首先我们分别对两个数据流使用朴素蓄水池算法进行取样，得到取样结果，分别表示对于总量为的数据流进行取样后所得取样结果，两个数组的长度均为（均远大于，且可以不相同）。因为程序实现时，可以对于数据流中元素的个数进行计数，所以均为已知的。

在合并两个取样结果时，为了保证合并后的取样结果等价于将数据流合并为，并使用朴素蓄水池算法进行取样后得到的取样结果，就需要在合并时，以一定的权重从随机抽取元素。权重取决于其所来自的数据流中元素的总数，即和。

合并时，从取样结果取出共个元素组成取样结果。定义概率，每次取元素时，我们随机生成一个取值范围在的浮点数（相当于掷一个硬币），如果，则从中不放回地随机取出一个元素放入，否则从中不放回地随机取出一个元素放入，时间复杂度为。此外，对于取样结果，数据流中的元素总数为。

最终，可以从两个数据流的个样本中抽出的个样本，且时间复杂度为，空间复杂度仅为。 (11)

合并两个取样结果的算法如下所示：

Algorithm 3‑2 合并取样结果的算法

|  |  |
| --- | --- |
| Algorithm 合并取样结果的算法 | |
|  | struct SampleResult{ |
|  | Array[k] samples; |
|  | int total; |
|  | } |
|  | function merge (SampleResult N, SampleResult M) { |
|  | SampleResult ret; |
|  | float P = N.total / (M.total + N.total); |
|  | for (int i = 0; i < k; i++) { |
|  | coin = generate a random float in [0, 1]; |
|  | if (coin >= P) |
|  | randomly move an item from N.samples to ret.samples; |
|  | else |
|  | randomly move an item from M.samples to ret.samples; |
|  | } |
|  | ret.total = N.total + M.total; |
|  | return ret; |
|  | } |

### 合并算法证明

取样结果中的元素是由朴素蓄水池抽样算法得到的。则数据流的每个元素被取样的概率分别为，而从中取出一个元素的概率是。所以，合并取样结果时，从数据流中取样一个元素的概率是。同理，从数据流中取样一个元素的概率也是。因此来自数据流的元素均有的概率被取样。 (11)

### 协调线程之间的合作

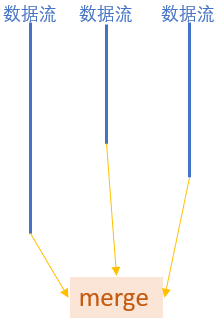
拥有合并算法后，想要将蓄水池算法推广到多个数据流非常简单：假设我们一共有台机器，分别对个数据流进行取样后，将个取样结果两两合并，一共合并次，就可以得到针对多个数据流的取样结果。其中针对数据流的取样可以并行执行，一个数据流由一个**取样线程**负责；而合并也可以由多个**合并线程**并行执行，继而就可以说整个算法是一个并行的算法。

现在，我们需要对于取样线程与合并线程进行管理，在享受多线程性能的同时，保证算法的正确性。存在以下几个问题：

1. 如何为合并线程分配任务（通过消息传递、阻塞队列还是异步编程）
2. 怎样交付取样结果（通过共享内存还是消息传递）
3. 由谁来交付取样结果并开始合并过程，什么时候可以开始合并过程

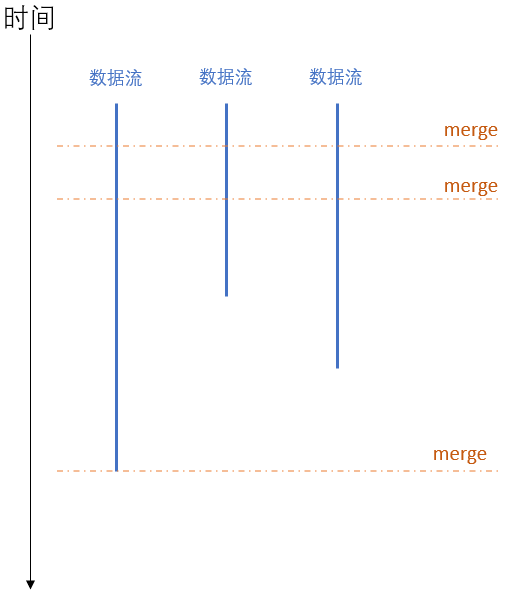
其中，对于问题1、2，我们会在第4章、第5章分别根据Java、Rust的语言特性给予各自的解决方式。简单地说，Rust和Java提供了不同的并发编程范式来解决这些问题：Rust和Java都支持了基于OS线程的多线程开发；此外，Rust还支持消息传递和async/await异步编程。

对于问题3，存在两种选择。第一种选择是：由取样线程交付取样结果，在数据流没有更多元素时立刻交付。这么做的好处是，在某些数据流取样完成后，我们可以立刻开始进行合并。取样过程与合并过程之间没有间隔。这可以以最快的速度完成整个取样算法。此方法由于不需要解决数据竞争的问题，所以不必为使用锁而付出代价。



图片 3‑2由取样线程交付取样结果

第二种选择是，由用户（调用者）来决定何时交付取样结果。用户在某个时间点调用函数，收集各个数据流的取样结果并将它们合并。这样做的好处在于，可以根据当前已经收到的数据多次生成取样结果，符合实际生产中数据流没有穷尽的情况。



图片 3‑3由调用者决定何时交付取样结果

但是这种方法的缺点在于：不能在某些数据流已经结束时，就立刻开始对从这些数据流中得到的抽样结果进行合并，这势必导致效率不如第一种方法高。

如果对两种办法同时进行实现，必然会导致代码的复杂度直线上升。经过考虑，笔者决定根据第二种方法组织代码。

@todo

# 基于Java的蓄水池算法实现

本章给出了一种并行蓄水池算法的实现方式，并使用Java对并行蓄水池算法进行了代码实现。

## 数据结构

为了实现并行蓄水池算法，我们需要首先实现朴素蓄水池算法。SimpleReservoir类专门用来实现了朴素的蓄水池算法。

对于每个采样线程，他们都在进行朴素蓄水池算法抽样，但我们不能直接使用SimpleReservoir类，因为在交付结果时存在数据竞争的问题。因此我们需要继承了SimpleReservoir类的SamplerHandle类，在实现了朴素蓄水池算法的同时，使用锁帮助我们解决数据竞争的问题。

ParallelReservoir类包含了并行蓄水池算法所需要的合并算法，承担着实现并行蓄水池算法的任务。在需要获得各取样线程的结果并将它们合并为一个取样结果时，只需要调用ParallelReservoir::getSampleResult()即可。

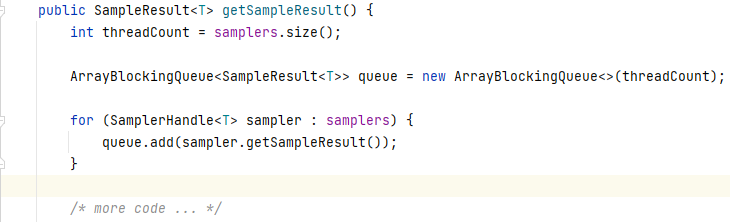
SampleResult即取样结果，是数据类，包含了两个域：样本（数组）和数据流中元素总数（整数）。

## 算法实现

经过前文（3.2.3协调线程之间的合作）的讨论，我们选择由使用者决定何时交付取样结果。基于此，我们需要讨论如何传递取样结果，并协调合并线程。

### 传递取样结果

在前文（3.2.3协调线程之间的合作）中我们提到需要由使用者决定何时交付取样结果，所以在主线程调用ParallelReservoir::getSampleResult()后，每个取样线程的取样结果由该方法放入阻塞队列，并启动合并线程。如下图所示：



图片 4‑1ParallelReservoir::getSampleResult()部分代码截图

所以会有两个线程同时访问SamplerHandle所在的内存：取样线程调用SamplerHandle类的trySample(T element)方法进行取样；而主线程的ParallelReservoir会调用SamplerHandle类的getSampleResult()方法取得该取样线程的取样结果。（具体可见图片 4‑5描述SamplerHandle与ParallelReservoir之间的合作关系的UML时序图）所以此处存在数据竞争的情况（data racing）。

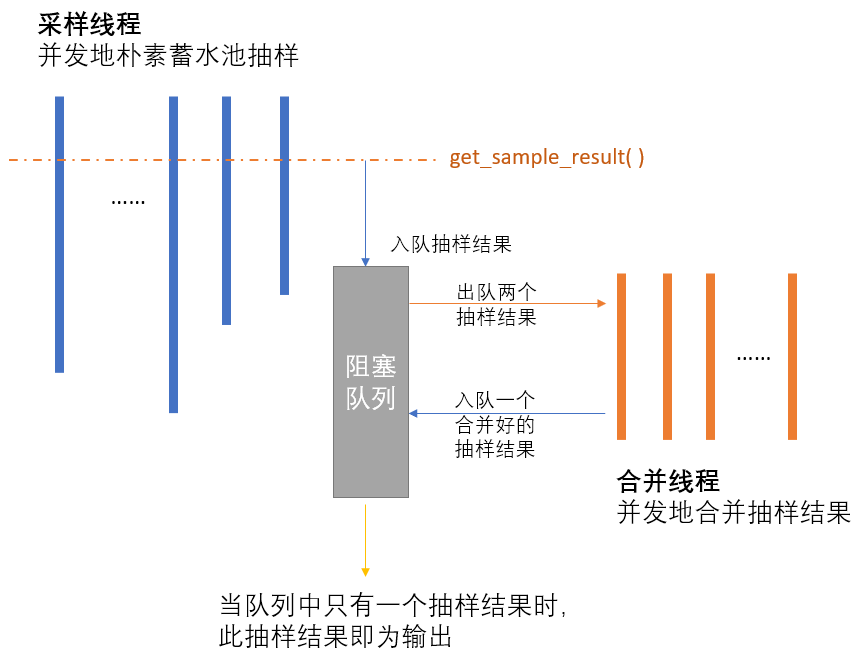
因此，有必要为SamplerHandle添加锁，以防止两个线程同时对其访问时导致的数据竞争（data racing）。实际上，SamplerHandle就是原子化的SimpleReservoir，它将SimpleReservoir的每一个都封装了获取锁与释放锁的过程。

### 协调合并线程

Java支持基于OS线程的多线程编程，可以创建若干线程来执行任务。此外Java标准库中提供了一些线程安全的集合，方便程序员安全优雅地管理线程，而不是使用最基础的锁来管理线程。

这里我们可以通过标准库中提供的**阻塞队列**（blocking queue）来管理线程。阻塞队列是一种通过提供锁来保护共享的数据结构。他的本质是一个队列（queue），生产者线程向队列中插入元素，消费者线程则获取元素。使用阻塞队列可以安全地从一个线程向另一个线程传递数据。

当试图向阻塞队列添加元素（put）而队列已满，或是想从队列移出元素（take）而队列为空的时候，阻塞队列会导致线程阻塞。 (12)



图片 4‑2利用阻塞队列实现并行蓄水池算法的示意图

如图片 4‑2所示，现有个线程并发地执行朴素的蓄水池抽样，并将取样结果添加至阻塞队列中（put），我们称这些线程为**取样线程**（蓝色），他们均是阻塞队列的生产者。

同时，另有若干**合并线程**（橙色），他们总是试着从阻塞队列中移出（take）两个取样结果，并利用前文所述的合并算法将这两个取样结果合并为一个。合并完成后，将取样结果放回合并线程既是阻塞队列的生产者，也是其消费者。

判断算法结束的标准是：当队列中只剩下一个取样结果，且所有的取样线程、合并线程都结束自己的工作时，并行蓄水池算法结束。这个标准比较难以执行，好在如果我们每次只将两个取样结果合并为一个，就一定只有次合并。据此可以简单地判断算法是否结束。

倘若每个合并线程只进行一次合并（取出两个取样结果，放回一个，线程销毁），我们可以很轻易地判断合并是否完成——只需要判断是否进行了次合并即可；而程序上的实现也非常简单——只需要开个合并线程，将他们全部交给阻塞队列管理，然后主线程对于这些合并线程的句柄（handle）调用join()函数即可（即让主线程阻塞，等待所有合并线程完成工作后再继续执行）。

但是，创建个线程进行合并的开销是比较大的，尤其考虑到合并算法的复杂度仅为，而取样数量一般是一个比较小的数字，所以事实上每个合并线程创建后，没有进行太多计算就被销毁了。由此想到，可以模仿异步编程中任务的概念来解决管理合并线程、判断算法结束的问题。

每一次合并被看作一个任务。一个合并线程可以进行多次合并，即完成多个任务。当一个任务完成后，线程不销毁，而立刻进行下一个任务。要管理这些合并线程也很简单，我们创建一个以整数为类型的原子量（Atomic Variable）a，由所有合并线程共享，且初始化为0。每个合并线程不断检查a是否小于。若小于，则从队列中取出两个取样结果进行合并；否则线程终止。合并线程每完成一次合并，就将a加一。这样即可完成对合并线程的管理。这种方法可以在一定程度上降低创建与销毁OS线程的开销对于算法执行速度的影响，但问题在于创建多少个合并线程比较合适。

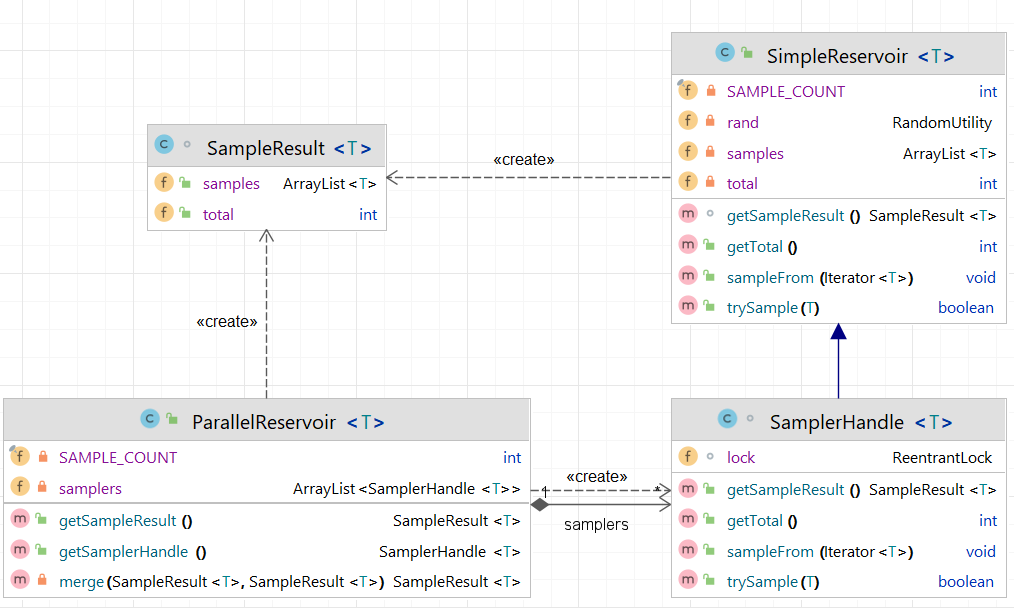
在代码实现中，笔者采用了第一种方法，即“每个合并线程只进行一次合并”。在此给出伪代码：

Algorithm 4‑1使用阻塞队列实现的并行蓄水池算法

|  |  |
| --- | --- |
| Algorithm 使用阻塞队列实现的并行蓄水池算法 | |
|  | BlockingQueue<SampleResult> queue; |
|  | for (sub\_stream s: sub\_streams) do in PARALLEL { |
|  | sequential reservoir sampling; |
|  | put one SampleResult into queue; |
|  | } |
|  | for (int j = 1; j <= sub\_streams.size() – 1; j++) |
|  | create merging thread with behavior{ |
|  | SampleResult r1 = take one SampleResult from queue; |
|  | SampleResult r2 = take one SampleResult from queue; |
|  | SampleResult mergeResult = merge(r1, r2); |
|  | put mergeResult back to queue; |
|  | } |
|  | } |
|  | for (int j = 1; j = sub\_streams.size() – 1; j++) |
|  | join merging thread[j] |
|  | SampleResult finalResult = take one SampleResult from queue; |
|  | return finalResult |

## 代码组织

Java代码的UML类图如下图所示。



图片 4‑3由IntelliJ生成的Java代码的UML图

### SimpleReservoir<T>类

代码中，SimpleReservoir<T>类实现了朴素蓄水池算法，它有两个主要方法。第一个是boolean trySample(T element)，负责取样传递给类的元素，并将是否取样以布尔值的方式返回。第二个重要方法是SampleResult<T> getSampleResult()，他负责返回取样结果。

表格 4‑1SimpleReservoir重要的成员函数与成员变量

|  |  |
| --- | --- |
| boolean trySample(T element) | 取样传递给方法元素，并将是否取样以布尔值返回。算法伪代码见3.1朴素蓄水池算法 |
| SampleResult<T> getSampleResult() | 返回取样结果 |
| int total | 计数器，统计数据流中元素的总数 |
| ArrayList<T> samples | 数组，表示目前为止算法得到的样本 |

### SampleResult<T>类

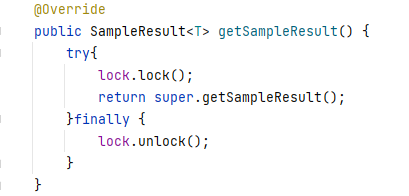
SampleResult<T>类是一个数据类，也就是我们之前所谈论的“**取样结果**”。其成员变量有两个，第一个域是集合类，负责存储**样本**；第二个域是一个整形，负责存储目前为止数据流中元素的总数。

表格 4‑2SampleResult<T>类的成员变量

|  |  |
| --- | --- |
| int total | 计数器，统计数据流中元素的总数 |
| ArrayList<T> samples | 数组，表示目前为止算法得到的样本 |

### SamplerHandle<T>类

SamplerHandle<T>类继承了SimpleReservoir<T>。它只有一个新的成员变量：一个重入锁（reentrant lock），用来保护自己不会因为多个线程的同时访问而出现数据的不一致性。SamplerHandle重写了SimpleReservoir的几乎每一个方法，使得对父类的方法进行访问时，需要先调用锁的lock()方法。代码如下图所示：



图片 4‑4SamplerHandle覆写SimpleReservoir的成员函数的示例

### ParallelReservoir<T>类

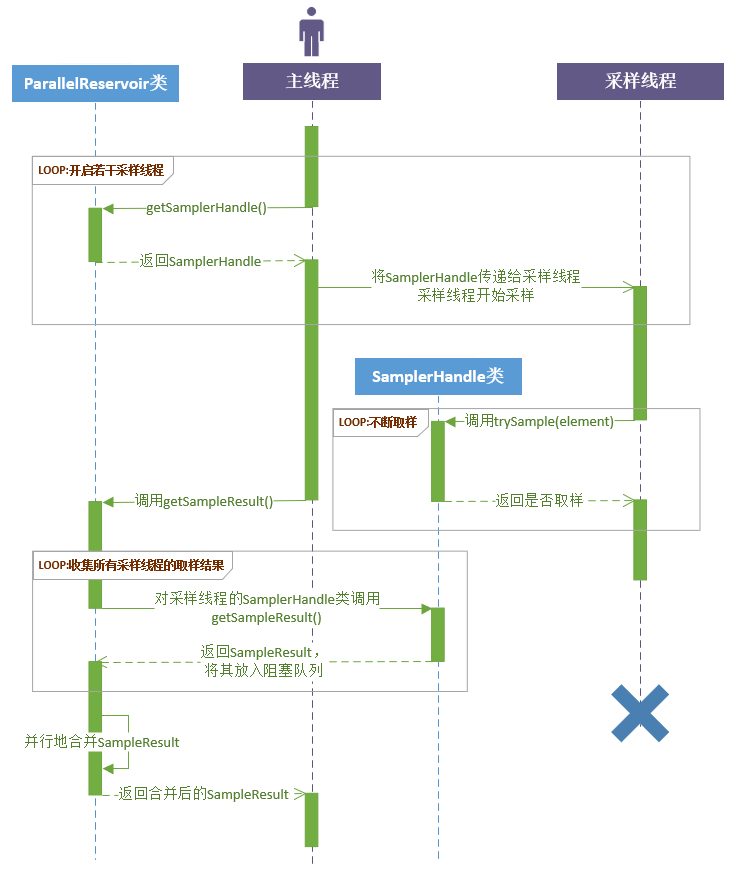
ParallelReservoir实现了并行蓄水池算法，其重要成员见下表。

表格 4‑3 ParallelReservoir类的成员方法和成员变量

|  |  |
| --- | --- |
| SamplerHandle getSamplerHandle() | 生成一个SamplerHandle类，交由主线程传递给取样线程 |
| SampleResult getSampleResult() | 向取样线程收集取样结果，将他们并行地合并后返回 |
| SampleResult merge(SampleResult a, SampleResult b) | 这是一个私有方法供合并线程调用，可以将两个取样结果合并为一个 |
| ArrayList<SamplerHandle> samplers | 存储指向各线程持有的SamplerHandle的引用，以便收集取样结果 |

ParallelReservoir类不会创建取样线程，它们需要由数据结构的使用者创建。在getSampleResult()方法向取样线程收集取样结果，将结果放入阻塞队列中，并创建个合并线程将他们合并（其中为取样线程的个数，即samplers.size()），在等待所有合并线程结束后，将阻塞队列中最后的SampleResult返回。

为了让读者更好地理解ParallelReservoir类与SamplerHandle类的合作关系，笔者绘制了UML时序图，以供参考：



图片 4‑5描述SamplerHandle与ParallelReservoir之间的合作关系的UML时序图

# 基于Rust的蓄水池算法实现

笔者在前一章使用Java实现了蓄水池算法。现将根据之前的经验，通过Rust实现了蓄水池算法。Rust的语言特性导致其对于算法的实现与Java有很多区别，这些区别将在本章一一讨论。

此外，本章对比了Java和Rust代码的执行性能；并为代码进行了正确性验证。

## 数据结构

Rust和Java采用了类似的数据结构，包括用于实现朴素蓄水池算法的SimpleReservoir，数据类SampleResult以及实现了并行蓄水池算法的ParallelReservoir。不同之处在于，Rust的Mutex可以帮助我们完成SamplerHandle的任务，直接对SimpleReservoir进行封装，保证不会发生数据竞争的问题。因此，在Rust中，Mutex<SimpleReservoir>替代了SamplerHandle。

关于省去SamplerHandle类的具体内容，本文会在5.3.2代码复用一节进行讨论。

此外，由于Rust的命名规则与Java不同，所有的方法和变量由驼峰命名改为下划线命名。

## 算法实现

Rust同样面临3.2.3节所提到的两个问题：传递取样结果和协调合并线程。但Rust给出了不同的解决方式：基于Rust类型系统的Mutex和消息传递。

### 传递取样结果

由于一些原因（见5.3.2节），在Rust中不再像Java代码一样使用SamplerHandle封装SimpleReservoir。在代码实现中，我们使主线程、取样线程共享SimpleReservoir，在主线程需要取样线程的取样结果时，调用共享的SimpleReservoir::get\_sample\_result()。这是一种通过共享内存进行协作方式。为了保证多个线程访问SimpleReservoir时不会影响数据一致性，笔者使用了Mutex封装了SimpleReservoir，即使用Mutex<SimpleReservoir>，这和Java代码中使用SamplerHandle，借助Reentrant Lock封装SimpleReservoir是类似的。

Rust的优越之处在于，如果我们没有为SimpleReservoir添加Mutex封装，而直接在线程间共享SimpleReservoir，Rust的编译器会立刻发现其中的问题，并在编译期就组织代码运行，警告我们存在数据竞争、所有权等问题，并可能提供一些好用的解决方案。并且，由于Rust强制程序员使用RAII范式，所以Mutex锁会在锁离开作用域时自动释放，避免了很多时候程序员会忘记释放锁的情况。

Go语言中有一句经典名言：“Do not communicate by sharing memory; instead, share memory by communicating”。Rust的开发者除了可以通过共享内存实现线程间的协作，还可以在通过通道（channel）线程之间传递消息，帮助线程协作。

此时Rust也需要SamplerHandle对SimpleReservoir进行封装和扩展，以便使其获得消息传递所需要的函数

我们可以不再令主线程和同时拥有SamplerHandle的所有权。让取样线程拥有SamplerHandle。在主线程需要取样线程的取样结果时，向取样线程发送请求，取样线程通过通道向主线程返回取样结果。或者为SamplerHandle添加finish()方法。取样线程在数据流结束时调用此方法，通过该方法实现通过通道将取样结果传递给主线程。

如果使用消息传递，还需要解决一些问题。在收集取样结果时，取样线程可能已经结束，也可能没有结束。如何（多次）收集这些已经结束了的采样线程的采样结果是一个问题。

### 协调合并线程

在Java代码的合并过程中，我们使用阻塞队列协调主线程、合并线程之间的合作。

但是，Rust标准库并没有给我们提供阻塞队列（blocking queue）这一并发数据结构，第三方库中有基于异步通道实现的阻塞队列的包。但这里我们可以直接使用异步通道代替阻塞队列。由于mpsc通道只有一个接收者（即接收者没有Clone特征，无法调用clone()），所以需要在创建合并线程之前，由主线程从通道中取出两个取样结果，再由合并线程拿走这两个取样结果的所有权。（见图片 5‑1在主线程中取出通道中的取样结果，再传递给合并线程）



图片 5‑1在主线程中取出通道中的取样结果，再传递给合并线程

通道可以有多个发送者，所以合并线程在完成合并后，可以通过tx\_for\_merger将合并结果放入通道。

## Rust的编程范式

### 并发编程范式

Rust提供了基于OS线程的多线程编程和async/await异步编程两种并发编程范式。我们需要一种合适的范式来实现蓄水池算法。

本文需要实现的蓄水池算法是一种长时间运行的CPU密集型任务，需要CPU在某个线程持续运行；相比之下，async并发模型可以在一个任务处于IO或者阻塞状态时及时切换到另一个任务，任务切换的开销远小于线程切换的开销，async并发模型更适合服务器场景下的高并发。

事实上, async 底层也是基于线程实现，但是它基于线程封装了一个运行时，可以将多个任务映射到少量线程上，然后将线程切换变成了任务切换，后者仅仅是内存中的访问，因此要高效的多。 (2)

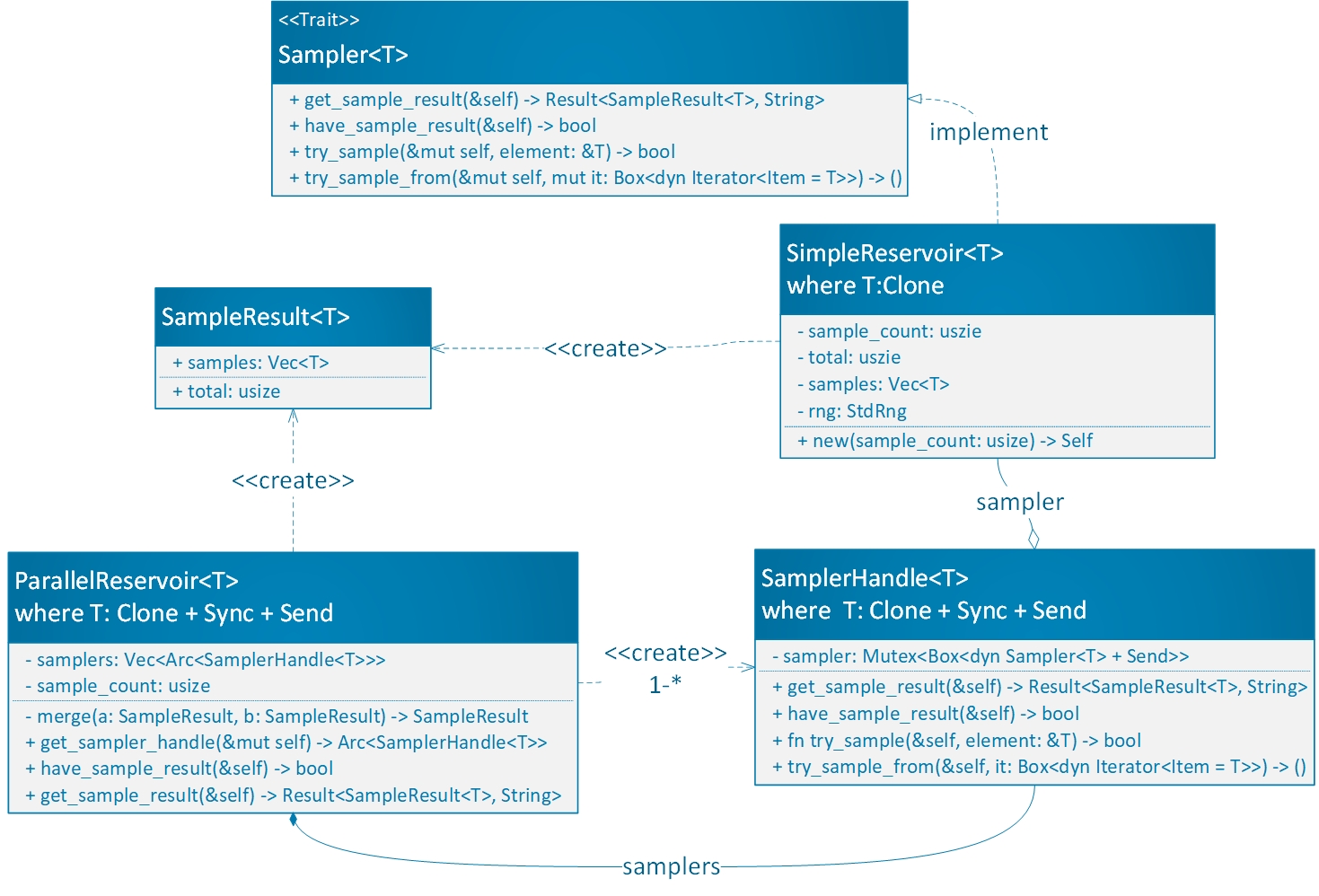
同时，Rust的异步编程暂时存在着一些兼容性问题，并且因为async和社区开发的运行时依旧在不断革新而有着更高的维护成本。当然，Rust也有成熟的异步编程的第三方库：tokio。

可以预想到在实际的生产环境中，取样过程应该是有很多I/O或是阻塞发生，应该使用async异步编程范式进行开发。但是在本文的实验中，我们的数据流来自于一个自增的整数（实际是来自Rust的迭代器），因而不存在I/O或是阻塞，反而成为了CPU密集型任务。同时，为了在同一层面对比Rust和Java的性能，我们选用基于OS线程的并发编程范式进行Rust代码的开发。

### 代码复用

最初构建Rust代码时，笔者试图模仿Java代码的结构，添加了Sampler<T>特征，用来描述实现串行蓄水池算法所需的所有API。再由SimpleReservoir类将其实现。

并行蓄水池算法不直接使用SimpleReservoir，而使用SamplerHandle类来解决数据竞争问题，其唯一成员是由Mutex封装的特征对象Box<dyn Sampler<T> + Send>。特征对象Trait Object曾在2.2.2节有所介绍。使用特征对象而不直接使用SimpleReservoir是为了以后可以方便地替换为别的取样算法，包括滑动窗口和加权蓄水池采样算法等。 (13)



图片 5‑2模仿Java代码结构写出的Rust代码的UML类图

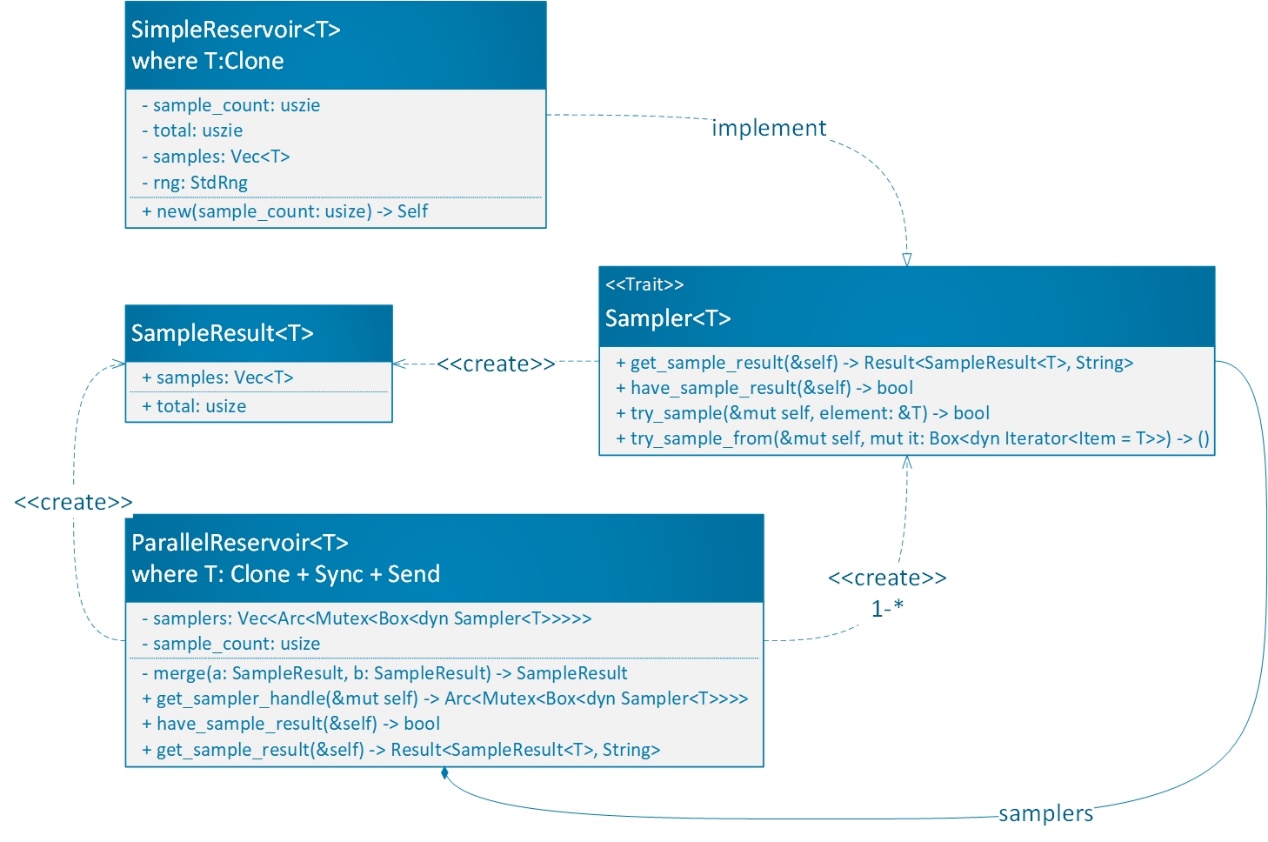
SamplerHandle仅仅是一个可以被多线程同时访问的SimpleReservoir，是为解决多线程中获取和释放锁的问题而存在。最初笔者希望为它实现Sampler特征，就像Java代码中SamplerHandle直接继承了SimpleReservoir一样。

但是这样设计没有充足的理由，并且由于Rust的在所有权上的限制，不方便为SamplerHandle实现Sampler特征，原因如下：

1. SimpleReservoir类和Sampler特征中的try\_sample(&mut self, element: &T)方法对于self是可变借用。
2. SamplerHandle是我们在主线程创建的，我们需要将他其所有权分享到取样线程。这时需要用Arc封装SamplerHandle（Arc是指Atomic Reference Count，可以让多个变量分享一个内存资源的所有权)。形如Arc<SamplerHandle<T>>。
3. 但Arc不允许内部量是可变的。如果要实现多线程内部可变性，考虑到SamplerHandle是会被多个线程访问的，且需要在线程之间传递（需要Send特性），所以应当用Mutex对其进行封装。形如Arc<Mutex<SamplerHandle<T>>>
4. 然而SamplerHandle就是为了用Mutex封装内部的SimpleReservoir而存在的，这样反复用Mutex封装是不合理的。
5. 有没有可能将Sampler::try\_sample方法对于self的借用改为不可变的？我们可以使用RefCell再对SimpleReservoir的成员进行封装，但这太不优雅了，且有3个成员需要封装。

综上所述，由于Rust语言的限制，笔者没有办法根据Java代码的设计思路在Rust中实现继承关系。

但是，笔者注意到由于Rust语言中Mutex封装的便利性，似乎可以将SamplerHandle这个对Box<dyn Sampler<T>>的封装去掉。即在ParallelReservoir中使用Arc<Mutex<Box<dyn Sampler<T> + Send>>>。但这可能使我们没有办法重构代码，通过使用消息队列让线程更优雅地合作（见5.2.1和5.2.2节）。



图片 5‑3 省去SamplerHandle对于Mutex<Sampler<T>>的封装后的UML类图

同时，这么做会将锁的获取与释放的工作留给使用ParallelReservoir的程序员。由于Rust的互斥锁可以在离开作用域时被释放（也可以手动drop），所以测试代码（即使用者需要写的代码）并没有增加太多。但ParallelReservoir的代码由180行左右缩减到了120行左右。

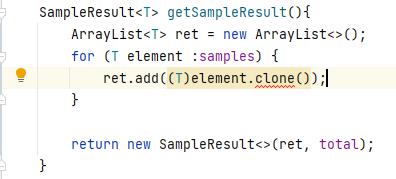
经过测试，这些改动对性能没有太大影响。但在5.4.3节中，我们将会讨论使用特征对象带来的性能波动。

### 类型系统与特性约束

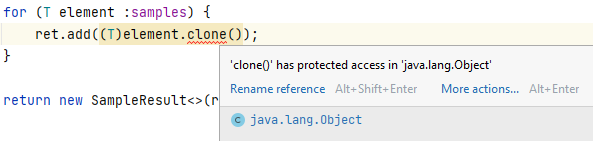
Java那章中，可以注意到所有的类后面都有一个泛型标记：<T>（T代表某种类型）。这意味着这些类都是泛型类，他们可以取样任意类型的数据，而不仅仅是int类型。Java可以对泛型进行约束，比如要求T是某种类型的父类或子类，或是实现了某个接口等，但无法对这个类型提出一些更为具体的要求，比如要求重写了Object.clone()函数，但这在Rust中可以做到。

#### Clone特性

在SimpleReservoir::getSampleResult()中，我们需要将存储在成员变量samples中的样本复制到SampleResult中，并将它们返回。可由于Object.clone()是一个protected的方法，需要我们对其进行覆写才能使用。而在Java中，我们没有办法要求泛型对于clone()进行了覆写。



图片 5‑4Java中使用clone()的代码



图片 5‑5没有针对泛型的约束所引发的错误

所以最终，我们只能使用ArrayList的clone()方法，对样本进行浅拷贝。

相比之下，Rust可以对于泛型进行特征约束，可以仅为实现了某些特征的泛型实现方法。在Rust中，我们可以要求泛型T实现了Clone特性，此特性确保泛型T具有clone()函数，方便我们复制元素。

#### Sync和Send特征

Sync和Send特征是Rust语言安全并发的重中之重。他们均为标记特征（marker trait），没有定义任何行为。

二者的功能在于：实现Send特征的类型可以在线程之间安全地传递其所有权；实现Sync特征的类型可以(通过引用)在线程间安全地共享。这两个特征仅仅作为标记存在，并没有定义任何行为。

一个由拥有Send特征的类型组成的结构体（struct）也会被编译器自动加上Send特征；类似，如果一个结构体由拥有Sync特征的类型组成，那么它也拥有Sync特征。Rust的基础类型一般都拥有Send和Sync特征，而手动为某种类型标记Send和Sync特征是不安全的（Unsafe），因为这会绕过编译器的检查。 (1)

（以下讨论中，SamplerHandle并没有被删去）

SamplerHandle是由主线程生产，并传递给取样线程的类（见图片 4‑5描述SamplerHandle与ParallelReservoir之间的合作关系的UML时序图）。

Java中，我们使用Lambda表达式创建合并线程，并捕获SamplerHandle。Java编译器对于这种跨线程的变量捕获不会太为难我们。然而对于Rust语言，在线程之间传递变量的所有权时，需要经过编译器严格的检查：被传递者的类型必须要有Send特性。

考虑到SamplerHandle所做的就是封装了一个Mutex。Mutex有Sync和Send特征，所以SamplerHandle也应该拥有Sync和Send特征。但当我们为了抽象，SamplerHandle.sampler的类型由SimpleReservoir<T>修改为特征对象Box<dyn Sampler<T>>后（dyn Sampler<T>是一个实现了Sampler特征的对象，称为特征对象，由一个智能指针Box进行封装），SamplerHandle不再默认实现Sync和Send了。没有Send特征，也就无法跨线程传递其所有权。

解决方法有二：

一是手动为SamplerHandle添加Send和Sync特征；但是这是Unsafe代码，Rust编译器无法帮助我们检查其中的问题，也不够优雅。

二是将sampler的类型修改为Box<dyn Sampler<T> + Send>。如此Box封装的特征对象有Send特征后，编译器就将自动依次为：

* Box<dyn Sampler<T> + Send>
* Mutex<Box<dyn Sampler<T> + Send>>
* struct SamplerHandle<T> { sampler: Mutex<Box<dyn Sampler<T> + Send>>, }
* Arc<SamplerHandle<T>>

实现Send特征，并允许我们在线程之间传递SamplerHandle。

### Rust的错误传递

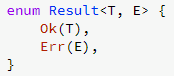
蓄水池算法需要有错误传递。该算法的功能是：从长度为N的数据流中取出k个样本。那么如果数据流尚未产生k个元素时就要求算法返回取样结果，就会产生错误，笔者称这种错误为“数据不足错误”。

在Java中，我们在遇到错误时返回null。然而null的发明者Tony Hoare曾经说过一段非常有名的话：

我称之为我十亿美元的错误。当时，我在使用一个面向对象语言设计第一个综合性的面向引用的类型系统。我的目标是通过编译器的自动检查来保证所有引用的使用都应该是绝对安全的。不过在设计过程中，我未能抵抗住诱惑，引入了空引用的概念，因为它非常容易实现。就是因为这个决策，引发了无数错误、漏洞和系统崩溃，在之后的四十多年中造成了数十亿美元的苦痛和伤害。

有鉴于此，Rust吸取了众多教训，决定抛弃null，而改为使用Option、Result这两个枚举变量来表示null、表示处理的错误与否。 (2)

Result<T, E>是一个枚举类型，定义如下：

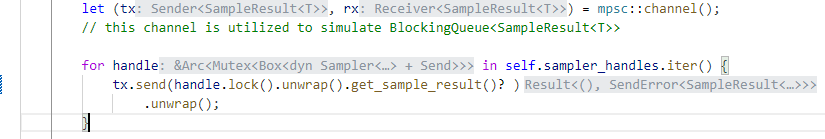


图片 5‑6枚举类型Result的定义

泛型参数 T 代表成功时存入的正确值的类型，存放方式是 Ok(T)，E 代表错误时存入的错误类型，存放方式是 Err(E)。

SimpleReservoir::get\_sample\_result()方法的返回值就是Result< SampleResult<T>, String>，如果数据流中元素的个数N大于k，则返回Ok(SampleResult<T>)，否则返回Err("Not enough elements in one or more Sampling Thread(s)".to\_string())。

而在并行蓄水池算法中，如果一个取样线程出现了数据不足，那么ParallelReservoir::get\_sample\_result()也应该返回Err。所以，好在Rust提供了?运算符以供我们方便地传递错误。?运算符可以在Result的类型是Err时，直接返回一个Result::Err。

‘

图片 5‑7 使用?传递错误

## Rust与Java代码的性能对比

@todo

### 测试平台与方法

测试平台具体信息如下：

表格 5‑1测试平台信息

|  |  |
| --- | --- |
| CPU | intel i7-13700KF @3.4GHz，8P+8E，24线程 |
| 主板 | MS icraft-Z690 |
| 内存 | DDR5 32G 6000MHz C30 |
| 操作系统 | Windows 11 Pro 22H2 |
| rust complier版本 | 1.68.0 |
| JDK版本 | 19 |

在Rust测试中，使用第三方benchmark包Criterion进行自动测试；在Java中我们使用自己编写的benchmark程序进行测试。

测试数据量（单次测试）如下：

表格 5‑2单次测试的数据量

|  |  |  |
| --- | --- | --- |
|  | ParallelReservoir | SimpleReservoir |
| 取样数量k | 1000 | |
| 单个取样线程的数据量N | 1000\_0000 | 1\_6000\_0000 |
| 取样线程个数 | 16 | 1 |
| 总数据量 | 1\_6000\_0000 | 1\_6000\_0000 |

测试方法是：运行100次单次测试，对于收集到的100个运行时间进行整理后，得出单次测试的运行时间。

### 性能测试结果

表格 5‑3benchmark结果

|  |  |  |
| --- | --- | --- |
|  | ParallelReservoir | SimpleReservoir |
| Java | 362.58ms | 2256.18ms |
| Rust | 265.71ms | 1631.5ms |

表格 5‑4归一化后的benchmark结果

|  |  |  |
| --- | --- | --- |
|  | ParallelReservoir | SimpleReservoir |
| Java | 1.36 | 1.38 |
| Rust | 1.00 | 1.00 |

就测试结果来看，Rust的性能并没有超出Java太多；并且并行计算中，Rust与Java的差距变得更小了。不管怎样，在蓄水池算法中Rust和Java所用时间的比值均大于Pereira, Rui等给出的1.05:1.89，也就是说，在别的项目中，Rust有比Java更强的性能表现。

### 使用特征对象带来的性能波动

在2.2.2节中，笔者介绍了特征对象。SamplerHandle的唯一成员变量是一个特征对象：Box<dyn Sampler<T> + Send>。这个特征对象的类型实际上是SimpleReservoir<T>。在对ParallelReservoir的测试中，笔者发现性能会出现波动。注意性能波动不是指单纯的性能损失，而是指测试结果在两个值之间反复变化。复现这种波动很容易，只需要多执行几次cargo bench命令即可。两次测试之间，代码均无修改。而朴素蓄水池算法的测试没有出现性能波动。

性能波动以一次cargo bench为单位出现，一次cargo bench会运行一百次被测试函数。倘若某一次cargo bench出现波动，Criterion对于并行蓄水池算法测试函数运行时间的估计值会增加30%~120%。笔者注意到发生波动时，时间增加的幅度与采样线程的数量有相关性，但不能解释这种相关性（注意测试平台的CPU是16核心24线程）。

图表 5‑1反映了性能波动增加的时间与采样线程个数的关系。绘制该图时，笔者对于不同采样线程个数的情况，分别运行10次cargo bench，取其结果的最高值与最低值的比值减一（）。10次cargo bench中，大约会发生两次性能波动。

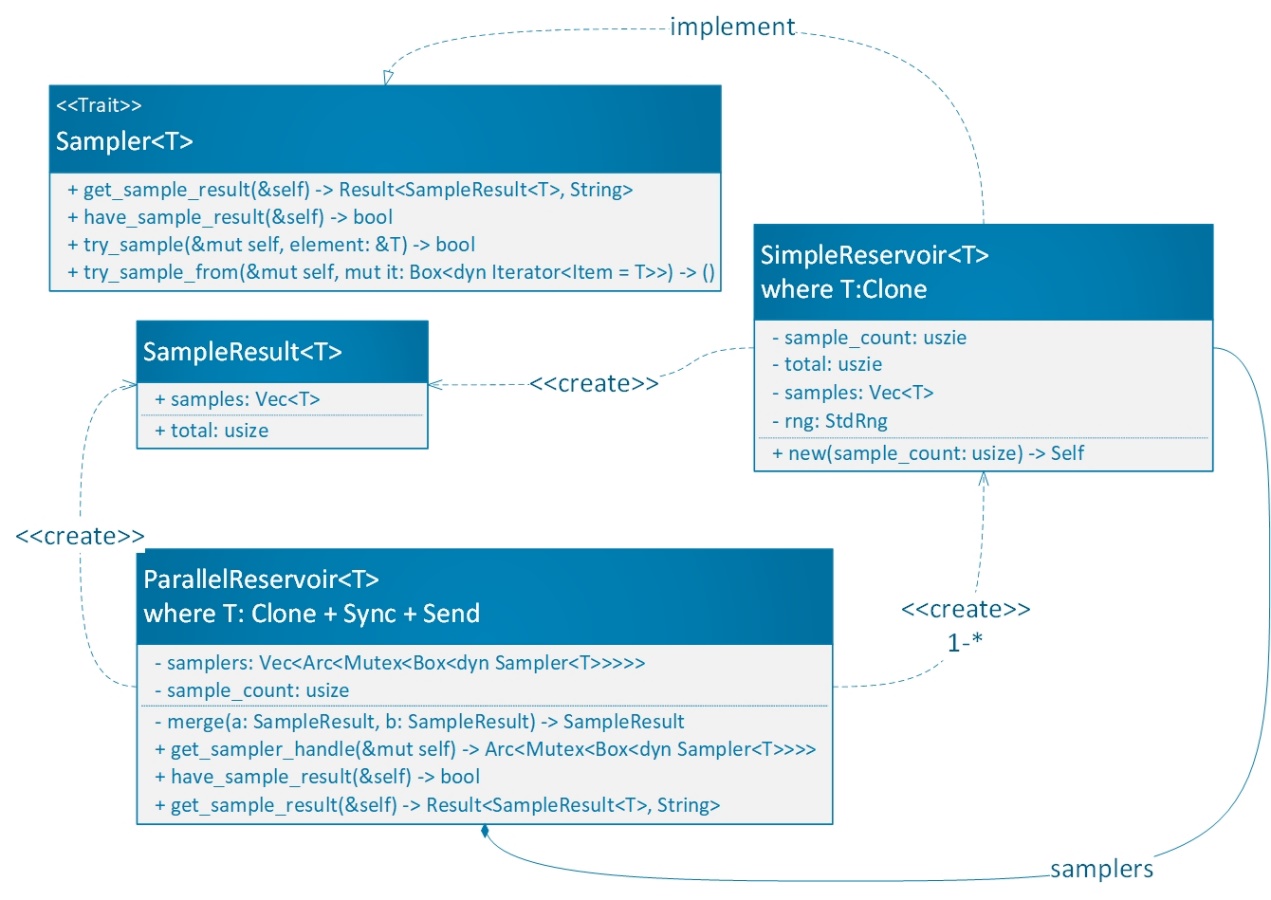
图表 5‑1性能波动增加的时间与采样线程个数的关系

此外，注意到在采样线程数比较小时（16个或更少），统计到的离群量很少。一次cargo bench过程中，一百个测试结果仅出现了5个左右的离群值；而在有更多采样线程时，则有较多的离群值出现，约15~20个。

为了找到波动的原因，笔者试着排除了可能的影响元素：关闭了CPU的节能模式，并保证测试中没有其他程序抢占CPU时间。并且排除了操作系统大小核心调度的问题，因为在另一个平台上测试时也出现了性能波动。（Ubuntu 22.10， intel i7-6700K，4核心8线程，虚拟机只使用其中4个线程）

因为能力有限，笔者最终没能确定导致性能波动的原因，但发现了消除波动的方法：将Mutex封装的内容由特征对象Box<dyn Sampler<T> + Send>修改为SimpleReservoir<T>。此后类似的波动不再出现。考虑到特征对象是一种动态分发（dynamic dispatch），其行为在运行时决定，因此使用特征对象可能会导致性能损失，但是不应该导致性能波动。

在弃用特征对象后，代码结构如“图片 5‑8 不再使用特征对象，性能波动消除后的UML类图”所示。

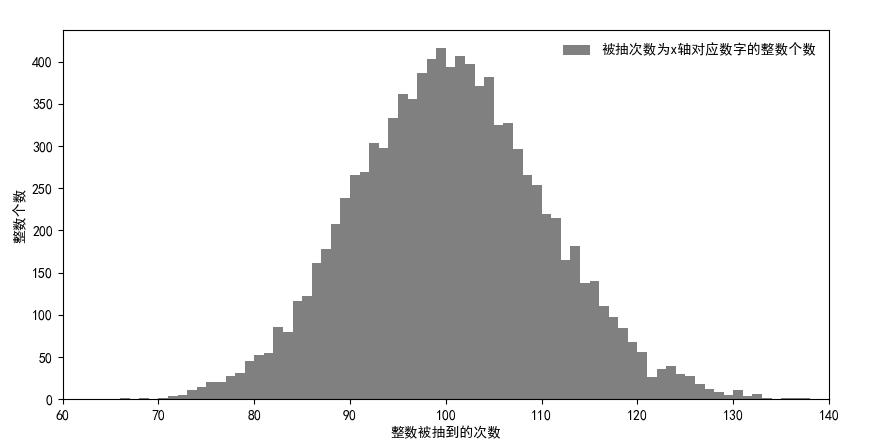


图片 5‑8 不再使用特征对象，性能波动消除后的UML类图

## 代码的正确性验证

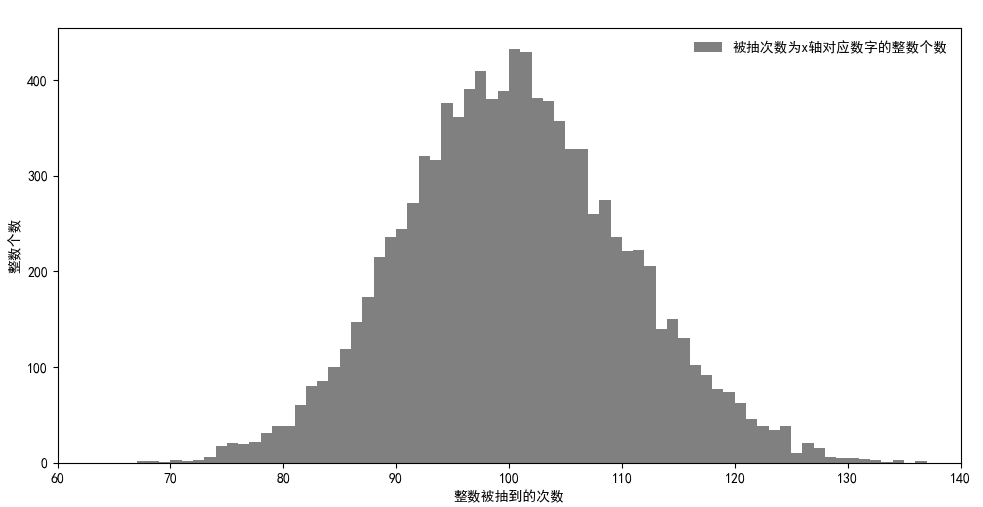
对于算法的形式化验证在第3章已经完成。现给出一种验证代码正确性的方法。假设执行一次代码时，数据流为1到的整数。程序从其中抽出个整数。执行次后，如果程序正确，则每个整数应该被抽到次左右。

现记录下每个整数被抽到的次数，以被抽到的次数为X轴，以（被抽次数为X轴对应数字的）整数个数为Y轴作直方图。看所得图像是否服从于平均值为100的正态分布即可。



图表 5‑2用Rust实现的SimpleReservoir的某次验证结果

对于ParallelReservoir代码（实现了并行蓄水池算法），在验证时将数据流均分为4份交由四个取样线程处理，最后再进行合并。



图表 5‑3用Rust实现的ParallelReservoir的某次验证结果

根据观察，图像基本符合正态分布图像的特征，由此可以间接地证明代码的正确性。

## 本章总结

本章最开始时直接仿照Java代码构建了Rust的代码（代码结构见图片 5‑2模仿Java代码结构写出的Rust代码的UML类图）。之后一步步从中发现问题并进行改动，最后的代码结构如图片 5‑8所示。这些改动包括：

1. 不再使用SamplerHandle进行封装，直接使用Arc<Mutex<SimpleReservoir<T>>>，同时放弃仿照Java的继承范式在Rust中进行代码复用。这样的做法可以完成本文的任务，但在代码重构时会要做更多工作。
2. 不再使用Box<Sampler<T> + Send>进行抽象，转而直接使用SimpleReservoir<T>，避免性能波动。
3. 使用Rust提供的Result枚举类型进行错误传递，而不是直接使用null

本章还有一些关于Rust语言本身的讨论，包括：

1. 使用Box<dyn Sampler<T>>而非Box<dyn Sampler<T> + Send>，进而导致代码中出现unsafe代码段的问题。
2. 使用消息传递（通道）部分替代阻塞队列
3. 使用特征约束对于类型进行额外的约束

# 总结

总而言之，Rust提供了所有权、生命周期和类型系统以供排查内存和并发问题。尽管Rust编译器对于代码的要求非常严格，令人难以上手；且较为熟悉Java、C++等拥有继承语言的程序员在Rust中需要改变自己的思维模式。但在熟悉Rust的要求后，程序员可以高效地开发出性能优秀且又安全的代码。

## **主要成果**

1. 对于并行蓄水池采样算法的实现方法进行了讨论，并给出了一种利用阻塞队列实现并行蓄水池算法的方法。
2. 利用Java和Rust分别实现了并行蓄水池算法
3. 根据对蓄水池算法的实现过程，对比了Java和Rust因为语言特性上的不同而导致两者在代码设计、程序性能上的区别。

## 未来工作

本文遗留了一些问题尚未解决：

1. 本文给出的代码尚不能在采样线程结束后就立刻进行取样结果的收集与合并。想要同时实现“采样线程结束后就立刻进行取样结果的收集与合并”和“使用者决定对于某个时间点的取样结果进行收集与合并”可能涉及到（包括但不限于）以下内容：  
   \* 给予各采样线程唯一的标记，清楚地了解每个线程的情况。  
   \* 使用消息传递重新设计线程间的协作方式
2. 尝试在Rust中以异步编程的并行程序编程范式实现并行蓄水池算法。异步编程与传统基于OS线程的多线程编程有很大区别，相互之间不容易复用代码。因为时间问题，本文没有使用异步编程实现算法。
3. 对使用特征对象带来的性能波动给出一个合理的解释。

参考文献

1. **NicholsCarol.** The Rust Programming Language. [联机] 2023年4月3日. The Rust Programming Language.

2. **Sunface.** Rust语言圣经. [联机] 2023年4月1日. course.rs.

3. **CramerTaylor.** Asynchronous Programming in Rust. [联机] 2023年4月3日. https://rust-lang.github.io/async-book/.

4. **Boqin Qin.** Understanding memory and thread safety practices and issues in real-world Rust programs. 2020年.

5. **秦伯钦.** 对现实Rust应用程序安全的实证研究. 2021年.

6. **Zemin Yu.** Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software. 2019年.

7. **牛聚川.** 融合模糊测试和形式化验证的RUST测试工具研究. 2022年.

8. **王烁程.** 面向新一代Intel可信执行环境的性能优化方案. 2021年.

9. **simonsan.** Anti-patterns in Rust, misuse of deref. Github. [联机] 2023年. https://github.com/rust-unofficial/patterns/blob/main/anti\_patterns/deref.md.

10. **PereiraRui.** Energy Efficiency across Programming Languages. Association for Computing Machinery. 2017年.

11. **BinsandBalls.** Distributed/Parallel Reservoir Sampling. Balls & Bins, A Mix of Random Thoughts. [联机] 2014年4月13日. https://ballsandbins.wordpress.com/2014/04/13/distributedparallel-reservoir-sampling/.

12. **HorstmannS.Cay.** Java核心技术 卷I 基础知识，第11版. 无出版地 : 机械工业出版社, 2019.

13. **MüllerAndré.** 4 Sampling Techniques for Efficient Stream Processing. Medium. [联机] 2022年3月1日. [引用日期: 2023年5月2日.] https://towardsdatascience.com/reservoir-sampling-for-efficient-stream-processing-97f47f85c11b.