

# 南京邮电大学

# 毕业设计（论文）

题 目      基于 Docker 的区块链基准测试与监控系统

---

专 业      物联网工程

---

学生姓名      阮秋帆

---

班级学号      B16070517

---

指导教师      王堃

---

指导单位      物联网学院

---

日期： 2020 年 2 月 17 日 至 2020 年 5 月 24 日

## 毕业设计（论文）原创性声明

本人郑重声明：所提交的毕业设计（论文），是本人在导师指导下，独立进行研究工作所取得的成果。除文中已注明引用的内容外，本毕业设计（论文）不包含任何其他个人或集体已经发表或撰写过的作品成果。对本研究做出过重要贡献的个人和集体，均已在文中以明确方式标明并表示了谢意。

论文作者签名：

日期： 年 月 日

## 摘 要

随着比特币、以太币等各种数字货币席卷全球，区块链进入了人们的视野。区块链以其去中心化、防篡改、可追溯等特性，使它不仅在支付和交易领域有着天然优势，而且也为供应链、存储、物联网、云计算等领域注入了新的活力，虽然它可以降低交易成本、简化存储架构、提高网络的安全性，但当前区块链的性能不足，使其应用也受到了限制。

因此，对于涌现的各种区块链架构，有必要对其进行性能分析，以期对它的不足进行改进。本设计在 Docker 环境下，提出了一个具有可移植性的区块链基准测试和监控的框架，主要应用数据库 Prometheus 以及可视化工具 Grafana 来完成对区块链性能指标的分析以及展示，在数据收集方面，不同区块链收集方式有所不同。本文首先对区块链背景及容器技术做出介绍，并分析了国内外相关研究，然后分析课题并完成系统的概要设计，最后详细描述系统构建所需的环境、软件，以及系统运转的流程，并展示了部分关键代码以及对功能的测试。

**关键词：**区块链；基准测试；性能；容器；监控

## ABSTRACT

As various crypto-currencies such as Bitcoin and Ethereum sweep across the globe, blockchain has entered our horizons. Blockchain with its decentralization, tamper-proof, traceability and other characteristics, not only has its natural advantages in the field of payment and transactions, but also bring new vigor and vitality into the fields of supply chain, storage, Internet of Things, cloud computing, etc. Although it works in reducing transaction costs, simplifying storage architecture, and improving network security, the current insufficient performance of blockchain has restricted its applications.

Therefore, it is necessary to analyze the performance of various emerging blockchain architectures in order to improve its deficiencies. In the Docker environment, this design proposes a portable framework of benchmark for testing and monitoring blockchain, which mainly uses Prometheus as a database, and Grafana which is the tool of visualization to construct an analysis-and-display framework of the blockchain performance, and the data collection is different for different blockchain. This article first introduces the background of the blockchain and related domestic or abroad research. Then analyzes the topics and presents the outline design of the system. Finally, it describes the operating environment, system operating processes required and releases some key code and functional tests.

**Key words:** blockchain; benchmark; performance; Docker; monitor

# 目 录

第一章 绪论 .....	1
1.1 区块链介绍 .....	1
1.2 DOCKER .....	2
1.3 设计动机 .....	3
1.4 章节安排 .....	3
第二章 国内外相关研究 .....	5
2.1 区块链性能瓶颈的研究 .....	5
2.2 区块链性能测试框架 .....	6
2.2.1 BLOCKBENCH .....	6
2.2.2 Caliper .....	7
2.3 本章小结 .....	7
第三章 需求分析与系统概要设计 .....	8
3.1 需求分析 .....	8
3.2 待测试区块链简介 .....	9
3.3 系统框架 .....	9
3.4 本章小结 .....	11
第四章 环境搭建及系统详细设计 .....	12
4.1 被监控机环境 .....	12
4.1.1 区块链和云服务器 .....	12
4.1.2 cadvisor 及其配置 .....	12
4.2 监控机环境 .....	14
4.2.1 系统环境 .....	14
4.2.2 Prometheus 及其配置 .....	16
4.2.3 Grafana 及其配置 .....	20
4.3 监控指标设计 .....	21
4.4 详细监控设计 .....	22
4.4.1 Fabric1.4 监控设计 .....	23
4.4.2 Ethereum 监控设计 .....	25
4.4.3 IOTA 监控设计 .....	27
4.5 本章小结 .....	28
第五章 测试与监控 .....	29

5.1 基准测试结果.....	29
5.2 实时监控结果.....	30
结束语 .....	36
致 谢 .....	37
参考文献 .....	38
附录 A 系统使用说明书.....	40

# 第一章 绪论

## 1.1 区块链介绍

区块链起源于比特币，世界对比特币的态度起起落落，但作为比特币底层技术之一的区块链技术日益受到重视。从 2019 年 10 月 24 日，在中央政治局第十八次集体学习时，习近平总书记强调，“把区块链作为核心技术自主创新的重要突破口”，“加快推动区块链技术和产业创新发展”。区块链已走进大众视野，成为社会的关注焦点<sup>[1]</sup>。

区块链本质上是由一组彼此不完全信任的节点维护的仅允许追加的数据结构，是一组串连的区块，每个区块包含很多交易，并通过哈希指针链接到前一个区块，其结构可见图 1.1。

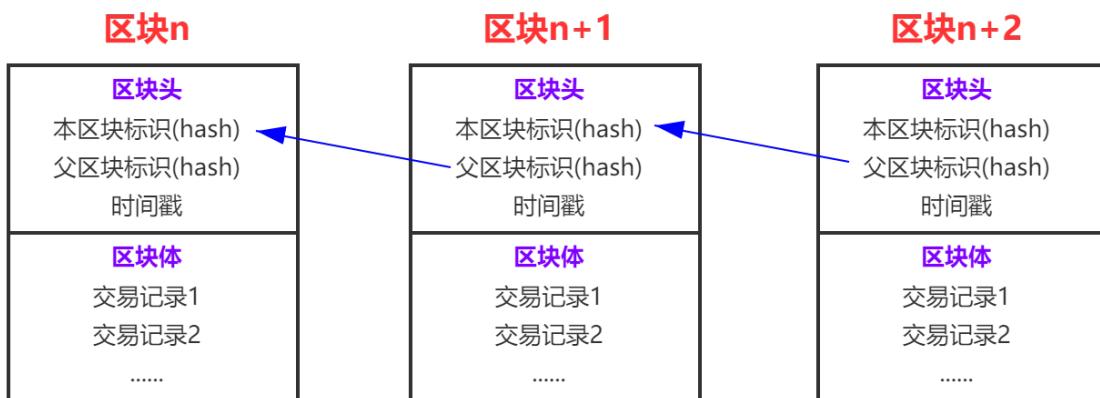


图 1.1 区块链链式结构

区块链核心技术如下<sup>[2]</sup>:

1) 分布式账本：在每个节点低位相同的前提下，区块链每个节点都按照块链式结构存储完整的数据，它们依靠共识机制保证存储的一致性。没有任何一个节点可以单独记录账本数据，从而避免了单一记账人被控制或者被贿赂而记假账的可能性。也由记账节点足够多，理论上讲除非所有的节点被破坏，否则账目就不会丢失，从而保证了账目数据的安全性。

2) 非对称加密：存储在区块链上的交易信息是公开的，但是账户身份信息是高度加密的，只有在数据拥有者授权的情况下才能访问到，从而保证了数据的安全和个人的隐私。

3) 哈希算法：区块链应用哈希（Hash）算法，通过事务中的数据以及上一个区块的信息，来产生下一个区块的 Hash 值，因为 Hash 具有单向性和唯一性，区块信息的变化将导致 Hash 值的变化，所以可以检测区块中的交易是否被篡改，确

保区块链数据的一致性和完整性，提高安全性。

4) 共识机制<sup>[3]</sup>: 是通过全网节点或特殊节点的投票，以“少数服从多数”的规则，完成对交易的验证和确认。对一笔交易，如果利益不相干的若干个节点能够达成共识，就可以认为全网对此也能够达成共识。目前区块链主要共识机制有工作量证明机制（PoW）、股权证明（PoS）、实用拜占庭容错算法（PBFT）等等。

5) 智能合约<sup>[4]</sup>: 智能合约基于可信且不可篡改的数据，执行代码，完成特定的功能，它包含了有关交易的所有信息，只有在满足要求后才会执行结果操作，所以允许用户在不需要第三方的情况下，执行可追溯、不可逆转和安全的交易。

区块链使用以上技术，实现了去中心化、防篡改、可追溯的特性。从应用的角度来看，它可以作为一个分布式数据库。通过共识机制的设计，它能够容忍拜占庭式失败，其透明性也降低了架构成本和人工成本，提供了比现有数据库系统更高的安全性，但区块链的机制使其吞吐量、时延、扩展性等性能状况受到严峻考验，在性能方面远不如现在中心化的数据库，它还有更大的发展空间。

## 1.2 DOCKER

随着 PaaS(platform as a service)的兴起，Docker 被给予了越来越多的关注。Docker<sup>[5]</sup>是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的镜像中，然后发布到任何流行的 Linux 或 Windows 机器上。Docker 的使用类似于常用的各种虚拟机，但他们有所区别，如图 1.2。同 VM 虚拟机不同的是，Docker 容器是在操作系统层面上的虚拟化<sup>[6]</sup>，而前者是在硬件层面的虚拟化<sup>[7]</sup>，因此，Docker 容器更加便捷小巧、启动快速，并且由于没有臃肿的从操作系统，容器运行时可以节省大量的磁盘空间、CPU 及内存。由此实现的高资源利用率，以及容器和容器、容器和宿主机的隔离，使得对 Docker 容器的监控可以有效测量应用程序的资源占用。同时，在 Docker 上进行开发、测试和部署也更加方便。Docker 的核心概念为镜像、容器以及仓库。镜像是一个独立对象，内部是一个精简的操作系统，并包含应用运行必须的依赖包和文件。容器则是运行起来的镜像，Docker 在镜像的最上层创建一个可写层作为容器。仓库则是存放镜像的地方，通过仓库可以方便地下载或是上传所需镜像。Docker 使应用的开发部署更加快速，更加便捷。

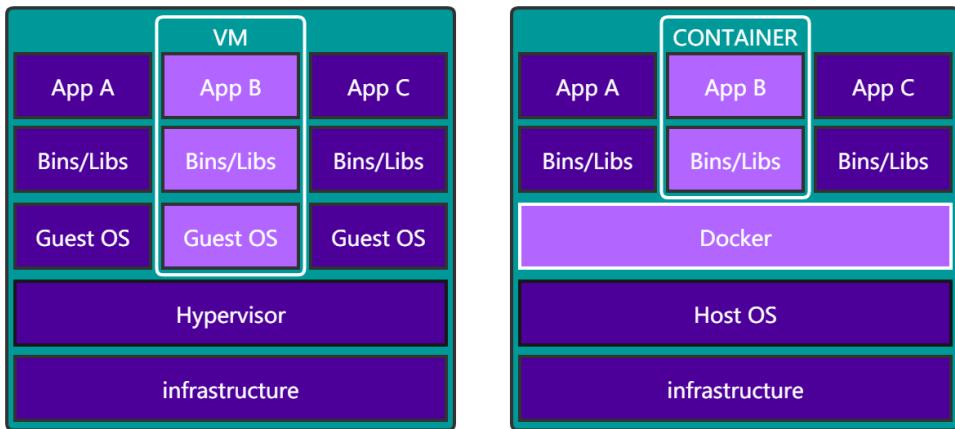


图 1.2 VM(左)与 Docker(右)与结构区别

### 1.3 设计动机

区块链虽然有着高可靠性、透明性等优势，但它本身是完全去中心化的 P2P 网络，受限于分布式的存储与信息交换方式，以及共识机制，区块链在吞吐量，延展性，资源利用率等性能指标方面明显低于中心化项目。随着数据量的增加，以及在应用过程中，受到实际环境的限制和任务对实时性、并发性等的硬性要求，区块链节点对资源的大量消耗，区块链交易的延迟性以及吞吐量都受到严峻考验。

目前各种区块链系统百花齐放，为了测试各种架构的实际效率，以便使用者了解该网络是否满足实际需求，便于开发者探求自身设计的区块链在某方面是否还有不足之处，也便于运维人员对其进行实时监控，本文设计了一个新的区块链测试系统。

由于主流的区块链项目采用了各不相同的设计理念和数据格式，基准测试系统必须能够拥有较好的兼容性。为此，系统采用模块化设计，分为数据收集、数据存储、数据可视化三个部分，并引入了 Docker 技术。使用容器技术运行区块链在部署上更加方便，环境不容易出错，而且通过对容器的监控来获取区块链系统对资源的占用情况。这套监控系统具有实时性、可移植性和稳定性，对于不同区块链系统的监控，只用更改数据收集部分。通过对不同区块链网络的基准测试以及性能比较，可以清楚地了解当今区块链系统在运行时的性能状况。

### 1.4 章节安排

本文第一章简要介绍了课题中的两个关键词：区块链和 Docker，提出了此次设计的优势和意义，并从整体上叙述了此区块链基准测试系统的结构。

第二章对国内外在区块链性能及测试框架方面的研究进行了概述，分析了现有技术的优缺点，吸收了部分研究的经验教训。

第三章在课题需求中，对设计的性能指标做了综述，并介绍了待测试的区块链系统，提出了本次设计的系统的框架。

第四章则说明了系统环境的搭建方法以及应用的详细配置，提出了系统的详细测试指标并针对不同区块链系统给出了具体的实现方法，

第五章给出了区块链系统的测试结果，并对数据进行了分析，同时展示了系统监控图。

在本文最后对所做的工作进行了总结，并提出了完善设计的思路和目标，表达了对未来研究的期望。

## 第二章 国内外相关研究

### 2.1 区块链性能瓶颈的研究

区块链的性能瓶颈是区块链系统面临的主要挑战<sup>[8]</sup>。一般用 TPS(每秒交易处理量)来表示吞吐量，如比特币，其 TPS 约为 7，并且当交易复杂的时候 TPS 还会下降，而信用卡每秒处理成千上万次交易，区块链的低性能制约着它在金融系统中需要高频交易的场景的应用。因此，许多设计中会使用区块扩容方案来提升区块链性能。本文通过对区块链进行量化分析，得出结论：增加区块链的块大小以及提高块生成速度都可以提升区块链性能，但大区块需要更快的网络速度，如果区块大小和 P2P 网络的传输速率不匹配，就会降低区块链性能。另外，在增加区块大小和块生成速率的同时，会使安全性降低，攻击者只需要更少算力就能够获胜。

区块链平台的性能是企业应用的一个主要关注点<sup>[9]</sup>，文献通过对 Hyperledger Fabric 的各个参数进行全面研究，分析其性能，找出了潜在的性能瓶颈。首先了解各种配置参数（如区块大小、认可策略、通道、资源分配、状态数据库选择）对事务吞吐量和延迟的影响，以提供配置这些参数的各种指导。并且发现背书策略验证、区块中事务的顺序策略验证和状态验证和提交（使用 CouchDB）是三个主要瓶颈，简单的优化方法有：在密码学组件中使用主动缓存进行背书策略验证（性能提高 3 倍）和并行背书策略验证（性能提高 7 倍）。

在性能研究中，经常将关系数据库与区块链进行比较，在 BLOCKBENCH<sup>[10]</sup>中，流行的两种数据库基准工作负载，分别为 YCSB 和 Small Bank，均被移植到所设计的区块链基准测试工具中，通过编写简单而有针对性的智能合约，来对其性能进行测试。实验选取流行的以太坊的 Go 实现，即 Geth v1.4.18 和 Parity 版本 v1.6.0，Hyperledger 版本为 v0.6.0-preview，在 48 个节点集群上实现，在基准测试中，Hyperledger 的性能始终优于 Ethereum 和 Parity，后两者在内存和磁盘使用方面产生大量开销，它们的执行引擎也比 Hyperledger 的效率低，但前者无法扩展到超过 16 个节点。Ethereum 和 Parity 对于节点故障更具弹性，但它们容易受到区块链分叉的安全攻击的影响。Hyperledger 和 Ethereum 的主要瓶颈是共识协议，但 Parity 的瓶颈是由交易签名引起的。Parity 将所有状态信息保存在内存中，因此它具有更好的 I/O 性能，但在硬件受限条件下无法处理大数据。而 Ethereum 仅将状态的一部分缓存在内存中（使用 LRU 策略），它牺牲吞吐量来换取更大的数据处理能力。文献<sup>[11][12]</sup>则深入到以太坊的操作码中，探求各操作码的 gas 消耗与系统资源消耗间的关系。

相较于无许可区块链系统比特币，目前出现许多许可区块链，它们通过降低共识算法的计算量和计算强度来提高吞吐量，文献<sup>[13]</sup>使用 Hyperledger Caliper 对许可

区块链 Hyperledger Sawtooth 1.0 进行性能测试，表明其吞吐量最高可以达到 2300tx/s，并且在输入事务数达到不断增长的时候，事务延迟和对主机资源的占用会成指数增长，超过最大吞吐量后，事务由于延迟将开始失败，这是由 Sawtooth 使用全部完成或全部失败的方式进行批处理的特性导致的，并且要想达到最大吞吐量，需要同步增加批处理的大小。除了吞吐量之外，区块链对系统资源<sup>[14]</sup>的占用以及安全性<sup>[15]</sup>也是其性能优劣的评判标准之一。

## 2.2 区块链性能测试框架

下文将介绍使用最广泛的两种区块链基准测试系统：BLOCKBENCH 和 Caliper，包括它们测试的方法、流程，以及优缺点。

### 2.2.1 BLOCKBENCH

不同区块链系统有着不同的架构，因此要设计一个通用基准测试系统十分困难，在 BLOCKBENCH<sup>[16]</sup>中，他们将区块链体系结构划分为三个模块化层：共识层、数据模型和执行层来分层研究。因为缺乏面向数据库的区块链工作负载研究，并且尚不清楚以太坊交易量这种工作负载是否足以代表评估区块链的一般数据处理能力，所以将区块链视为一个键值对存储结构，再加上一个可以通过智能合约实现交易和分析功能的引擎，成为基于真实和合成数据的事务和分析工具。BLOCKBENCH 分层结构如下图 2.1。



图 2.1 BLOCKBENCH 架构

在共识层上，BLOCKBENCH 使用一个名为 Do Nothing 的合约，接受事务作为输入并简单地返回，只涉及到极少执行层和数据模型层的操作，因此总体性能将由此衡量，即该项性能将直接由区块链事务延迟体现。

在数据模型层上，BLOCKBENCH 考虑区块链系统在进行历史数据的分析查询时的性能，这些查询由其数据模型决定。而 IO-Heavy 的评估，则是通过对合约

状态执行大量随机写入和随机读取来完成，可以通过事务的延迟来估计 I/O 带宽。

针对执行层，BLOCKBENCH 部署了一个智能合约来初始化一个大数组，并在上面运行快速排序算法。然后通过观察到的事务延迟来衡量执行层的性能。

BLOCKBENCH 存在两个主要问题：1) 虽然其测试所用任务集较为丰富，但是基准测试结构只涵盖系统吞吐量和时延两个性能指标，无法监控区块链网络对资源的使用情况，而资源相关的指标对于私有链的使用来说是非常有参考价值的。2) BLOCKBENCH 不具备实时读取测试结果的能力，其只能规定测试任务规模，然后在测试完成后采用脚本读取结果，这种方式降低了系统的可用性<sup>[17]</sup>。

## 2.2.2 Caliper

Caliper<sup>[18]</sup>是一种区块链性能测试框架，用户可以在定义好测试集的情况下对自己的区块链网络进行性能测试。当前支持 Fabric1.0、swatooth1.0 等区块链系统，实现了交易成功率、交易吞吐量 TPS、交易延迟、资源消耗四种指标。

Caliper 架构分为三层：1) Adaptation Layer(适配层)用途与编程中接口类似，用户定义的行为最终都会通过适配层，来对待测试区块链平台进行操作。2) Interface & Core Layer(接口及核心层)则包含资源监控、性能监控、报告生成模块等。3) Application Layer(应用层)用于定义区块链网络的配置，测试的相关配置，指 benchmark 和 network 两个文件夹的配置。

整个测试流程主要包括 3 个阶段：1) 准备阶段：用于初始化整个区块链网络，读取配置文件，部署智能合约，启动监控组建等。2) 测试阶段：根据定义好的 benchmark 配置文件，启动客户端子进程，执行相应的测试，返回统计结果。3) 报告阶段：分析统计结果，生成 HTML 报告。

Caliper 的优势在于它可以通过更改区块链网络的配置，从而达到丰富的测试效果，但它存在的不足则在于：1) 指标不够丰富，没有直观衡量区块链系统对资源的利用效率。2) 其可视化功能仍在开发当中。

## 2.3 本章小结

本章介绍了国内外对区块链性能的部分研究，并讨论了制约区块链性能的因素，现有技术条件下，区块链无法保证既安全又高效，因此还需要长足的发展。另外，本章也介绍了 BENCHMARK 和 Caliper 两个区块链测试框架以及各自的测试方法。在吸取前人的经验后，本文将在第三章对课题进行需求分析，从几个不同角度提出测试指标，并介绍系统的框架。

### 第三章 需求分析与系统概要设计

#### 3.1 需求分析

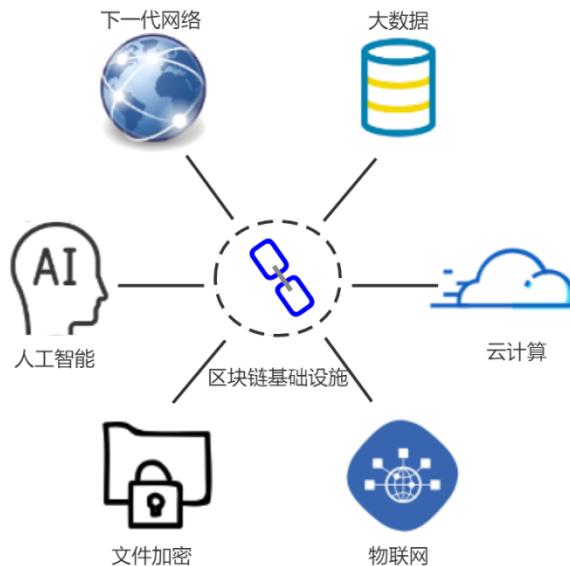


图 3.1 区块链结合其它技术

物联网、云计算等信息技术正处于高速发展的阶段，但二者在安全方面仍面临着巨大的挑战。物联网的容量和安全问题是制约其发展的主要因素<sup>[19]</sup>，而云计算在大规模应用中，用户隐私和安全性，以及数据完整性都是问题<sup>[20]</sup>，而区块链作为底层数据结构和技术形式，作为基础设施，它所带来的可靠性是物联网等领域发展所需要的<sup>[21]</sup>，正如图 3.1 所示。但在区块链应用<sup>[22]</sup>过程中，由于实际环境的限制，如节点资源的有限性，网络节点频繁地移动、加入和退出，都使区块链需要对自身的稳定性、可扩展性进行改进，同时在高并发和实时条件下，当今区块链的吞吐量及延时不足以应对市场需求。在区块链发展中，出现了许多架构，对于不同区块链系统的性能测试能够让用户和研究人员了解它们的特性以及发展瓶颈，这有助于未来的研究和应用。IOTA、Ethereum、Fabric1.4 是本次设计需要测试的区块链系统。

针对上述问题，本文将测试的性能指标分为了五个方面。

1) 可用性，即整体性能情况：吞吐量和时延，在云计算领域，大规模并行计算以及频繁的任务需求，包括大数据领域海量的数据处理任务，都需要区块链有足够的吞吐量来支撑上层应用，也需要较短的交易时延，防止事务堆积出现错误。

2) 资源利用率：包括区块链系统对 CPU、网络、内存、磁盘的资源使用情况。在资源有限的情况下，如物联网环境，节点的资源和能量是有限的，并且大部分能

量用于无线通信上，因此要高效利用系统资源。

本课题为“基于 Docker 的区块链基准测试与监控系统”，因此首先要在 Docker 上部署区块链，设计并获取区块链性能指标，并进行可视化处理。

需要监控的区块链包括 Fabric1.4, IOTA 以及 Ethereum，这些区块链有不同的设计架构以及对外接口。Fabric1.4 自带运维及管理功能，提供了可用于 Prometheus 使用的系统运行指标，包括事务延迟和产生的事务总数，而 IOTA 则需要使用 python 编写的可供 Prometheus 读取的客户端，将区块链的性能指标暴露在网页中，供数据库抓取。Ethereum 也需要使用 python 编写脚本来收集数据。三者的系统性能指标则可以统一使用 cadvisor 来监控。

为了将收集的数据按时间顺序展示，需要时序数据库 Prometheus 来存储这些数据，可视化的工作就由 Grafana 来完成。

### 3. 2 待测试区块链简介

1) Ethereum: 以太坊不仅有着自己的加密货币，而且提供了一个带有图灵完备语言的区块链，使用相应语言可以创建合约来编写任意状态转换功能。用户通过代码就能够创建一个基于区块链的应用程序，并应用于货币以外的场景，但以太坊采用 PoW 共识机制，这限制了它的扩展性以及交易确认的速度。Ethereum 不提供 metrics，因此需要编写 Prometheus\_client 来获取指标。

2) Fabric: Fabric 是超级帐本的项目成员之一，它也是一个分布式的智能合约平台。与以太坊不同的是，它在设计之初就是一个框架，而不是公有链，并且没有内置的代币。Fabric 将共识机制、身份验证等组件模块化<sup>[23]</sup>，使之在应用过程中可以方便地根据应用场景来选择相应的模块，并且 Fabric 采用容器技术，将智能合约代码（chaincode）放在 Docker 中运行，从而使智能合约可以用几乎任意的高级语言来编写。Fabric1.4 本身提供 metrics，可被数据库识别。

3) IOTA: IOTA 是新一代分布式账本，是完全为物联网结构设计的，它使用与以往区块链系统不同的架构：Tangle，一种基于有向无环图的新数据结构，它既没有区块，也没有链，但有着和传统区块链同样的规则。IOTA 通过 DAG 能实现较高的交易吞吐量，它通过平行验证完成共识，不用矿工传递信任，也不需要支付交易手续费，再加上它的可延展性，使得它在物联网、智慧城市领域有着天然优势。由于 IOTA 并不提供运维管理功能，因此监控时需要编写 prometheus\_client 来获取性能指标。

### 3. 3 系统框架

根据测试需求，系统不仅要能对区块链进行基准测试，也要能对性能状态变化

进行实时监控。前述的 BLOCKBENCH 需要通过编写智能合约来进行测试，但不同的区块链，如 Ethereum 主流使用 Solidity 来编写智能合约，而 Fabric 使用 go、java 等语言即可，虽然编程语言有相似性，但免不了要重新编写；Caliper 的测试结果会生成表格，展示的是静态数据，不够直观。它们都有各自的优缺点。

在本文所设计的监控框架内，既可以对静态数据测值，也可以通过图表展示性能变化，同时对于各种的区块链系统都有很好的适应性和兼容性。更重要的是，使用 Docker 来部署区块链以及监控系统<sup>[24]</sup>，不仅使应用的部署更加容易、使用更加便捷稳定，而且将区块链与其他应用隔离开来，减少外部环境对它的影响，使测试数据更准确。

测试系统框架主要分为以下三个部分：

### 1) 数据收集部分：

使用 cAdvisor 对容器指标进行监控。cAdvisor 是 Google 公司开发的监控工具，它为用户提供了对正在运行的容器的资源使用和性能特性的数据收集功能。它是一个守护进程，用于收集、聚合、处理和导出正在运行的容器的信息。具体来说，它为每个容器保存资源隔离参数、历史资源使用情况、完整历史资源使用直方图和网络统计信息。这些数据可通过基础查询界面查看，也可通过 http 接口，供 Prometheus 抓取。

对于区块链吞吐量等性能的基准测试，则通过区块链对外提供的接口，使用 python 或 go 编写的 Prometheus 客户端来提供数据，惊喜的是，Fabric1.4 直接对外提供 metrics，并且 IOTA 也有相应的 Prometheus 客户端，可以直接使用。

### 2) 数据持久化部分：

使用时序数据库 Prometheus，Prometheus 最初是在 SoundCloud 上构建的开源系统监控和警报工具，作为一个独立的开源项目，它在 2016 年加入了 Cloud Native Computing Foundation，成为继 Kubernetes 之后的第二个托管项目。Prometheus 的主要特征和优势为：①数据模型是多维的②具有灵活的查询语言 PromQL③默认通过 pull 方式主动抓取数据，也可以让节点通过 push 方式往数据库推送数据④支持静态配置和动态服务发现两种连接到目标服务器的方式。

Prometheus 的灵活性使得用户可以在服务器端对数据进行运算，减少对区块链系统的干扰，并且它可以同时监控多个端口，因此可以实现对多个区块链或服务器的监控，只要在容器内设置好端口映射即可。

3) 数据分析展示部分：在 Prometheus 基础上，本文使用 Grafana 来分析展示数据。Grafana 对于时序数据库 Prometheus 有良好的兼容性，图表的多样化和个性化设计也使监控系统更有实用性，它可以同时对设定好的多个性能指标进行监控。

由上述，系统框架如图 3.2 所示，数据收集部分使用 cAdvisor 获取服务器内运行的容器信息，使用 prometheus\_client 获取区块链信息，数据存储也通过 Prometheus 进行，最后通过 Grafana 对数据进行处理和可视化。

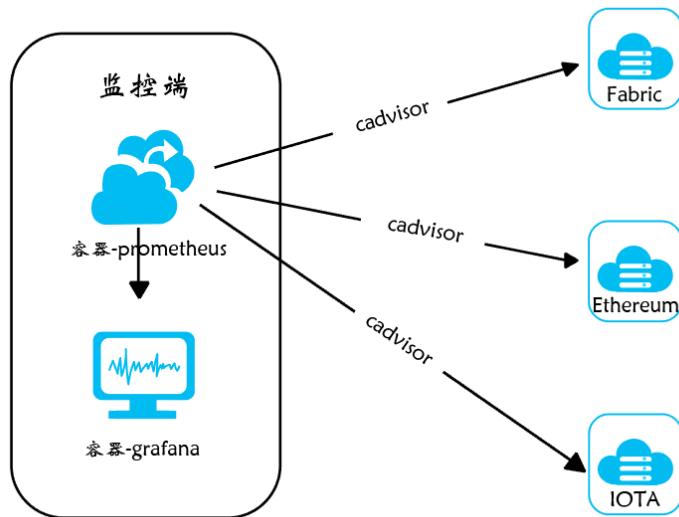


图 3.2 监控系统结构

### 3.4 本章小结

本章通过基准测试和监控需求进行分析，提出了几个测试的方面，并简要介绍了 Ethereum、Fabric 和 IOTA 三种区块链的特点，最后根据以上信息提出了具有较好的适应性和良好界面的基准测试系统，并介绍了其框架，为下一章节进行具体操作和详细设计做了铺垫。

## 第四章 环境搭建及系统详细设计

### 4.1 被监控机环境

由于需要对区块链进行测试并且保证各区块链处于相同的实验环境中，将其通过云服务器部署在 Docker 容器中。同时，为了获取容器数据，需要在被监控机上另外启动一个容器来运行 cadvisor。

#### 4.1.1 区块链和云服务器

区块链的测试和监控可以模拟实际应用的情况，将它部署在云服务器上，再对其进行操作，因此需要租用云服务器。在本文中选用的是阿里云服务器，规格如图 4.1 所示。使用 Xshell 软件新建连接，输入对应用户密码后即进入云服务器命令行界面，和虚拟机一样操作即可。在云服务器中需要安装 Docker 和 Go 环境以支持软件的运行，并且在安全组设置中开放端口，以使数据库可以远程访问客户端数据。

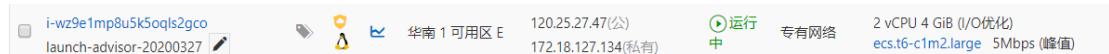


图 4.1 云服务器规格

在云服务器上安装 Docker 与在监控机上不同，由于需要部署区块链分布式节点，直接用 Docker 来手动启停服务的方式十分低效，因此云服务器需要使用 Docker-compose 来高效率部署节点。Docker-compose 是一个用于运行多容器的应用程序工具，可以通过编写 yml 文件的方式，将要处理的容器按文本顺序执行。使用 curl -L https://github.com/docker/compose/releases/download/1.25.4/docker-compose -`uname -s`-`uname -m` -o /usr/local/bin/docker-compose 来安装，使用 chmod +x /usr/local/bin/docker-compose 授予它可执行权限。Go 的安装可用 apt-get install golang-go 完成。

区块链的安装：①Fabric1.4：需要创建目录，并用 git clone <http://gerrit.hyperledger.org/r/Fabric> 来下载源码。②Ethereum：使用 docker pull ethereum/client-go 即可安装以太坊客户端的镜像。③IOTA：安装 maven 和 bazel，再克隆源码，详细操作可查看项目地址 <https://github.com/iotaledger/compass>。

#### 4.1.2 cadvisor 及其配置

本设计使用 cadvisor 作为数据收集部分，数据库通过其 http 接口来抓取容器数据，为了使 cadvisor 能获取 root 权限并访问到宿主机上其他容器的信息，需要

将系统文件挂载至容器中，因此启动时必须使用 volume 命令。主机的 /var/run 目录下存放正在运行的进程的进程文件，文件只有一行，为进程的进程号，/var/lib/docker/ 是 Docker 的默认文件目录，存放了镜像、容器，临时文件目录等信息，/dev/disk/ 下则存放了主机的磁盘信息，通过 --volume 命令将这些系统文件夹挂载至容器上，于是 cAdvisor 就能够访问到这些信息(不使用此命令的情况下容器与宿主机是隔离的)。每行 volume 命令最后的 ro 表示只读、rw 表示读写、wo 表示只写。

使用 Docker 部署 cAdvisor 的命令如下表 4.1：

表 4.1 Docker 运行 cAdvisor 的命令

```
docker run \
    --volume=/:/rootfs:ro \
    --volume=/var/run:/var/run:rw \
    --volume=/sys:/sys:ro \
    --volume=/var/lib/docker:/var/lib/docker:ro \
    --volume=/dev/disk:/dev/disk:ro \
    --publish=8080:8080 \
    --detach=true \
    --name=cadvisor \
    google/cadvisor
```

上述命令中，run 是 Docker 启动容器的命令，在启动后，Docker 会产生一个容器 ID，下次启动的时候可以使用 Docker start container ID 来启动上次停止的容器(保留了上次的设置和数据)，由于此命令会固定使用一个容器 ID，因此再次使用 cAdvisor 时需要使用 ID 来启动相应的容器，否则会提示容器 ID 已被占用。

由于 Docker 会实现容器与容器的隔离，因此在 Docker 中运行 web 应用时，要想访问到相应界面，应使用 --publish 命令来指定端口映射，使外部应用能够通过映射的端口访问到容器内应用提供的 web 界面，命令可简写为 -p。以 -p 10000:8080 为例，10000 为外部访问端口，8080 为 cAdvisor 提供的默认访问端口。在测试过程中，对于每个运行的区块链系统，若在不同的宿主机上，则宿主机需要各自运行 cAdvisor，并实现端口映射，以便监控端通过映射的端口将所有服务器的信息采集到一起。

cAdvisor 运行后，访问设置的端口，显示的界面如下图 4.2：

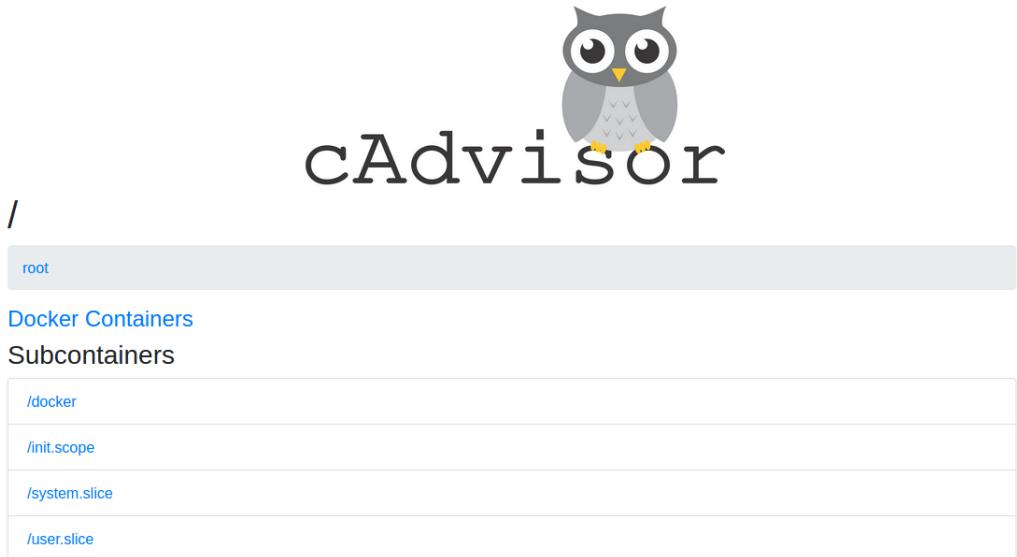


图 4.2 cAdvisor 主界面

选择 subcontainers 下的任意一个容器，可以看到其详细信息，如图 4.3 所示：

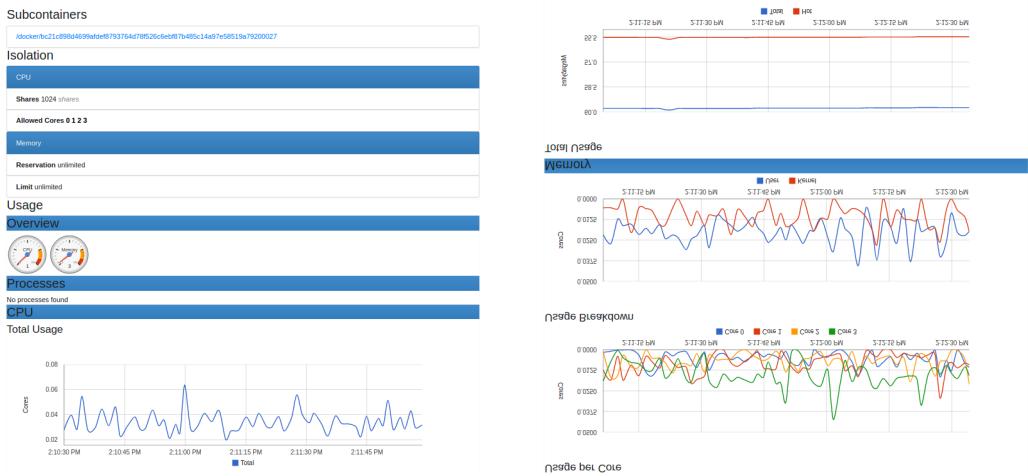


图 4.3 cAdvisor 查看某容器的资源使用情况

## 4.2 监控机环境

监控机环境包括系统环境，如 Python 依赖环境，Go 语言环境等，也包括系统模块的安装和配置，下文将具体介绍本系统所用到的各个配置。

### 4.2.1 系统环境

在本地环境下进行系统测试时，本文使用 VM VirtualBox 新建虚拟机，系统版本为 16.04.1-Ubuntu，GUN/Linux。在该版本的系统环境下，系统自带 python2.7 以及 python3.5，但系统默认 python 版本指向 python2.7，由于 python2.x 将停止支持，

所以需要将 python 版本转换为 3.x，以使代码有较好的适应性，不至在云服务器上无法运行，但同时也不可以删除 python2.7，因为系统中部分底层功能需要 python2 作为解释器。使用下列命令即可：①sudo cp /usr/bin/python /usr/bin/python\_bak，将 python2 进行备份，若之后出现错误或需要使用 python2 时，可重新将 python 命令指向此备份。②sudo rm /usr/bin/python，删除原先指向 python2 的文件。③sudo ln -s /usr/bin/python3.5 /usr/bin/python，使 python 默认指向 python3。

在编写数据库的客户端之前，需要安装对应的 prometheus.client 包，才可以建立数据库能够读取的指标类型。因此需要执行 sudo apt-get install python3-pip 安装 pip（一个通用的 python 包管理工具），然后使用 pip install prometheus.client 命令下载数据库 exporter 的库函数。

下一步需要安装 Docker。使用命令 sudo apt-get update 更新 apt 后，键入 sudo apt-get install -y docker.io，安装完成后，可以使用 docker version 来查看是否安装成功，注意在使用 Docker 命令前需要 su 来获取 root 权限。随后使用 systemctl status docker 来查看 Docker 服务是否运行，若是 inactive 状态，则需要 systemctl start docker 启动 Docker 服务，systemctl enable docker 命令则可以设置开机自启。

安装完 Docker 后，分别使用 docker pull prom/prometheus:latest、docker pull google/cadvisor:latest、docker pull grafana/grafana:latest 来下载它们的最新版本，

使用 docker run images 即可用默认设置运行对应镜像。

容器 ID 是更改容器设置所必须的参数，因此可以使用 docker ps 查看已经启动的容器的基本信息，但要查看包括已经停止的容器在内的所有容器，需要使用 docker ps -a 命令，如图 4.4 所示：

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
91bab278bd1e	grafana/grafana	"/run.sh"	7 days ago
Exited (255) 3 days ago	0.0.0.0:12200->3000/tcp	jovial_hermann	
8927dd13d5af	google/cadvisor	"/usr/bin/cadvisor -..."	9 days ago
Exited (0) 5 days ago		cadvisor	

图 4.4 查看容器 ID

实验完成后，需要将部分容器删除，以使系统环境和节点恢复成初始状态，使用 docker rm container ID 来删除单个容器，或者使用 docker container prune 对所有已经停止的容器进行批量删除。

其次，在上图 4.4 中 Grafana 的容器名称为系统随机分配的 jovial\_hermann 而不是 grafana，是因为启动时没有指定名称，因此启动时最好使用--name grafana 来指定名称。若容器已经启动，可以使用 docker rename jovial\_hermann grafana 将名称改为 grafana。在容器停止后，它并不会消失，而是保存下来，因此若要以相同设置和状态再次启动该应用，可使用 docker container ps -a 查看容器 ID，再使用 docker start container ID 启动。

需要注意的是，由于数据库在容器内运行，因此抓取指定 ip 地址和端口的数据时，可能需要使用 DNS 服务，然而容器内默认使用 127.0.0.1 作为域名服务器，所以使用域名访问宿主机可能会导致连接失败，因此需要配置 /etc/docker/daemon.json 文件，在里面加上{"dns": [ "10.0.2.15" ]}以添加宿主机 ip 地址

#### 4. 2. 2 Prometheus 及其配置

Prometheus<sup>[25]</sup>数据库是本次设计的核心，作为系统的时序数据存储部分。下面分四点来介绍 Prometheus 的基本信息以及详细配置。

1) 架构：

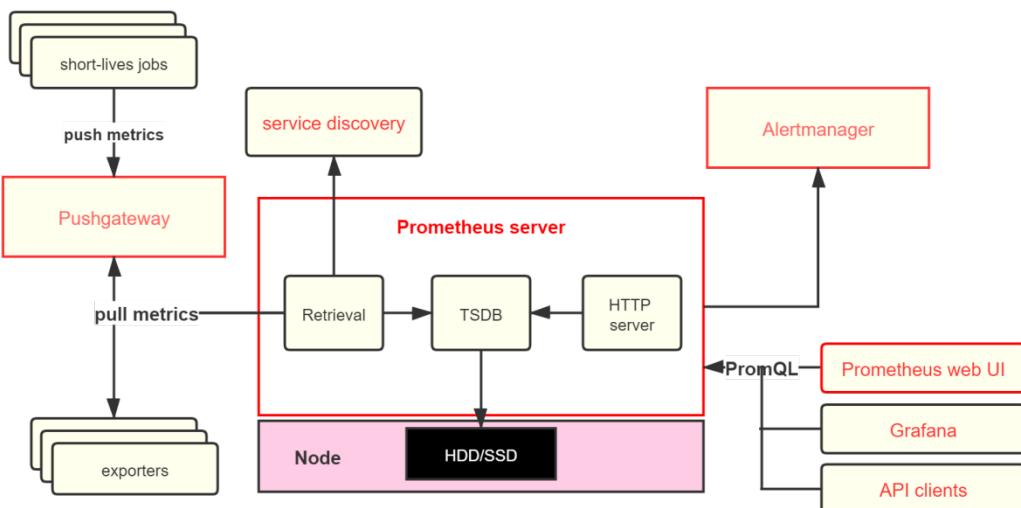


图 4.5 Prometheus 架构

根据架构图 4.5，Prometheus 通过配置文件静态添加监控节点或用服务发现功能来适应动态目标，并按照设置的时间间隔从节点拉取数据，默认的获取数据方式为 pull，也可以用 Pushgateway 方式，让不便于使用 pull 方式抓取的短时任务(可能数据库来不及抓取数据，任务便已经完成了)能主动将数据发送到数据库，随后将获取到的数据存入 TSDB，即时序数据库中，Prometheus 可以本地存储，优势是运维简单，但本地存储无法持久地获取和保存海量 metrics 数据，作为单节点存储的数据库，Prometheus 也提供远程存储的服务。在存入数据后，可以通过自带 WEB UI、Grafana 或 API 接口，使用内置的 PromQL 语言来查询所需数据。同时 Prometheus 自带报警功能，组件为 Alertmanager，但由于本系统使用了 Grafana，Grafana 中也有报警功能，使用起来更加方便，所以在 Prometheus 中不做赘述。

2) 指标(metrics)：

Prometheus 的数据模型称为指标(metrics)，采集到的样本点按时间顺序存储在

TSDB 中。指标实际上是一个多维数据，从整体上来看是一个 key-value 键值对，key 为<metric name>{<label name>=<label value>...}，value 则是测量值，同时在大括号中，可以定义多个标签名称和值，这同样也是一个键值对，多个标签使得 metrics 成为了多维数据，可以通过定义多个标签来区分指标，或是将特定的数据组织在一起。

### 3) 指标运算和查询：

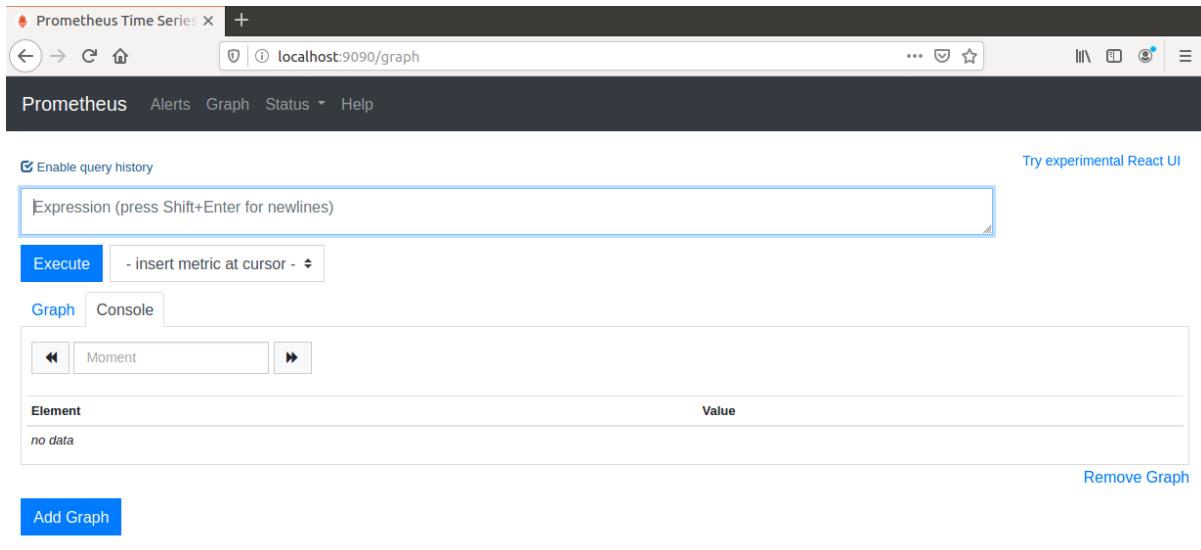


图 4.6 Prometheus 查询界面

PromQL 是 Prometheus 专用的查询语言，通过指标名称或标签来查询，Prometheus 会返回能够匹配的所有数据，返回的数据可以是瞬时数据，也可以是区间数据。如前述的 `http_request_total`，就是一个瞬时数据，表示此时总的 http 请求连接数，在瞬时数据后面加上`[time]`，则它成为区间数据，如 `http_request_total[1m]`，表示取一分钟内的数据，这组数据包含 1 分钟内的所有样本数据，若要查询 5 分钟前的数据，可以使用 `http_request_total offset 5m`，以及 `http_request_total[1m] offset 5m`。PromQL 的基本查询有：①查询键入名称和标签的所有数据：使用 `http_request_total{ip="192.168.168.168"}`。②模糊查询：`http_request_total{ip=~"192.*"}`，即 PromQL 支持正则表达式。③比较查询：`http_request_total>100`。

除了基本查询外，Prometheus 有内置函数以及运算法则，可以对选取的指标进行运算后输出。对于 Counter 数据有两种常用的函数运算：①`increase()`函数，如 `increase(http_request_total[1m])`，表示 1 分钟时间间隔上增加的 http 请求量②`rate()`函数用来求增长速率，`rate(http_request_total[1m])`表示每秒请求量，等同于 `increase(http_request_total[1m])/60`。注意到 PromQL 支持常用加减乘除符号的运算，但尽量不要对两个指标类型数据直接使用四则运算，否则会产生错误，无法获取到结果数据，例如标签不同的数据无法直接相加，查询界面为图 4.6。

### 4) 配置文件：

Prometheus 根据预先定义的配置文件来进行指标抓取，配置文件为 YAML 格式，有四个模块：global, alerting, rule\_files, scrape\_configs，分别为全局定义，警报规则设置、记录规则设置以及抓取配置。全局定义用来设置抓取间隔和规则(也可以在抓取配置下定义各自的抓取间隔)、警报计算的时间间隔，由于 Grafana 自带警报模块，所以在系统框架中有 Grafana 的情况下一般不使用 Prometheus 警报。

记录规则需要编写另一组 yml 文件，并添加到 rule\_files 模块下：1) 优势：用来预先在后台计算经常需要或占用较多资源的表达式，并将其结果保存为一组新的时间序列。因此，查询预先计算的结果通常比每次需要时执行原始表达式快得多。这对于仪表板十分有用，仪表板需要在每次刷新时重复查询相同的表达式，在大量查询时可能导致数据延时，从而使图表失真。2) 缺点：在规则文件中定义后，若需要修改表达式，则要对配置文件进行修改，并进行热更新，若文件编写出错，会导致数据库出错，因此使用 PromQL 来进行查询会更加灵活。

记录规则编写的 rule.yml 文件部分内容见表 4.2，其中 expr 为计算表达式，record 为新指标的名称。

表 4.2 rule.yml 文件

```

groups:
  - name: test-Fabric1.4
    rules:
      - expr: |
          sum(increase(broadcast_processed_count[5s]))/sum(increase(container_cpu_usage_seconds_total[5s]) *2.5)
          record: TPC_Fabric1.4

      - expr: |
          sum(increase(broadcast_processed_count[5s]))/(sum(increase(container_fs_reads_bytes_total[5s]))+sum(increase(container_fs_writes_bytes_total[5s]))) *1024
          record: TPIO_Fabric1.4

```

本次设计的 yml 配置文件部分内容如下表 4.3：

表 4.3 prometheus.yml 文件

```

global:                      #全局定义
  scrape_interval: 1s        #抓取数据的时间间隔
  evaluation_interval: 1s    #规则警报计算的时间间隔
rule_files:
  - "rule1.yml"

```

```

scrape_configs: #抓取配置
  - job_name: Blockchain-Fabric1.4 #定义标签 job 及名称 Fabric1.4
    static_configs:
      - targets: ['119.23.185.33:8443'] #配置抓取地址, 为目标 IP 的 8443 端口
      - targets: ['119.23.185.33:8080']
  - job_name: Blockcharin_Ethereum
    static_configs:
      - targets: ['49.235.69.130:8080']
      - targets: ['49.235.69.130:12345']
      - targets: ['49.235.69.130:12580']
  - job_name: Blockchain-IOTA
    static_configs:
      - targets: ['localhost:11111']

```

Prometheus 从 target 项抓取 metrics, target 配置最好写具体的 ip 地址, 而不是直接使用 localhost 或其他域名, 因为在容器中运行数据库的时候, localhost 会被 Docker 解析为环回测试地址, 从而导致连接 exporter 失败。在写好配置文件后, 可以使用 promtool check rules path 来检查语法错误, 确认无误后, 在启动数据库时, 应使用-v 选项, 将配置文件挂载到容器中, 如 docker run -name prometheus-p 9090:9090 -v /usr/local/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus, etc 路径为容器运行路径。

默认抓取的目标地址为 target 地址加上/metrics, 若使用浏览器, 可以通过 ip:port/metrics 访问到页面则表明可以成功抓取到数据, 协议为 http 协议, 若要更改这两者(如使用 https 服务的网页), 需要在 static\_configs 下添加表 4.4 中的设置。

表 4.4 配置文件其它设置

```

# 从目标获取指标的 http 资源路径.
[ metrics_path: <path> | default = /metrics ]
# 配置用于请求的协议方案.
[ scheme: <scheme> | default = http ]

```

监控数据存储在文件中, 在启动时需要自定义路径来持久化数据, 如表 4.5:

表 4.5 数据持久化设置

```

docker run -p 9090:9090 \
#tmp 为 linux 临时文件夹, 冒号前为自定义存储路径
-v /tmp/prometheus-data:/prometheus-data \
#数据存储时间, 此处为一天

```

```

-storage.local.retention 168h0m0s \
#保存的 chunk 最大大小，默认为 1G，即 1048576
-storage.local.memory-chunks=50502740 \
#数据库进行采集和查询数据的时候，需要大量全局锁，如果分配的 mutex 不够
    会导致数据延迟，因此在高并发采集数据时可以适当提高此项
-storage.local.num-fingerprint-mutexes=300960
prom/prometheus

```

数据库运行期间，若要更改配置文件，有两种方法进行不停服更新。一是热更新，二是通过服务发现来动态更新。服务发现可以使数据库动态增加减少监控节点，例如多个云服务器节点启动、关闭时动态添加监控端口，并且在配置文件过大时，可以实现配置文件的切分，或编写短配置文件动态添加进监控配置。

在本次设计中采用的是热更新方法。使用 vi /path 更改完配置文件后，在 Linux 命令行中使用 ps -ef | grep prom/prometheus 查询数据库的进程号，并使用 kill -HUP pid 即可完成热更新，如图 4.7 所示，在 Prometheus 的日志界面可以看到配置文件重新加载的信息。

```

level=info ts=2020-03-25T09:53:56.558Z caller=main.go:762 msg="Completed loading of configuration file" filename=/etc/prometheus/prometheus.yml
level=info ts=2020-03-25T09:53:56.558Z caller=main.go:617 msg="Server is ready to receive web requests."
level=info ts=2020-03-25T09:58:58.907Z caller=main.go:734 msg="Loading configuration file"
filename=/etc/prometheus/prometheus.yml
level=info ts=2020-03-25T09:58:58.914Z caller=main.go:762 msg="Completed loading of configuration file" filename=/etc/prometheus/prometheus.yml

```

图 4.7 配置文件加载成功

需要注意的是，yml 文件的编写必须要注意内容的缩进，否则会产生错误。上述步骤成功后，可以在 localhost:9090/targets 页面查看 exporter 的连接情况，up 为成功，down 为失败。

#### 4. 2. 3 Grafana 及其配置

Grafana 是开源的可视化平台，能够支持众多现有数据库，它主要用于展示大规模的时序数据。在本设计中它将获取 Prometheus 的数据并展示在其 web 界面上，它的默认端口为 3000，为了在宿主机上访问容器内的 web 页面，启动时可使用 12200:3000 来映射端口。

Grafana 对不同的数据源使用特定的查询语言都不同。以 Prometheus 作为数据源时，使用 PromQL 语言来查询和计算指标数据。Grafana 使用仪表盘(DashBoard)展示数据，仪表盘的基本组成单元为行(Row)，每行表示一种信息，如 CPU 利用率，一个仪表盘上可以添加许多行，以展示多个节点的性能。

Grafana 的仪表盘有许多种，在本设计中主要使用 Graph 或 Gauge<sup>[26]</sup>来展示 TPS、TPC 等实时指标，可以看到区块链系统对各项资源的实时占用，并且 Grafana 在 V4 以后，新增了报警功能，可以在仪表盘中设置阈值，当指标超过阈值，就会发出警报来提示。

使用 `docker run -d -p 12200:3000 grafana/grafana` 启动，然后访问 `localhost:12200` 即可，配置数据源为 Prometheus，并设置数据源接口，成功后界面如图 4.8：

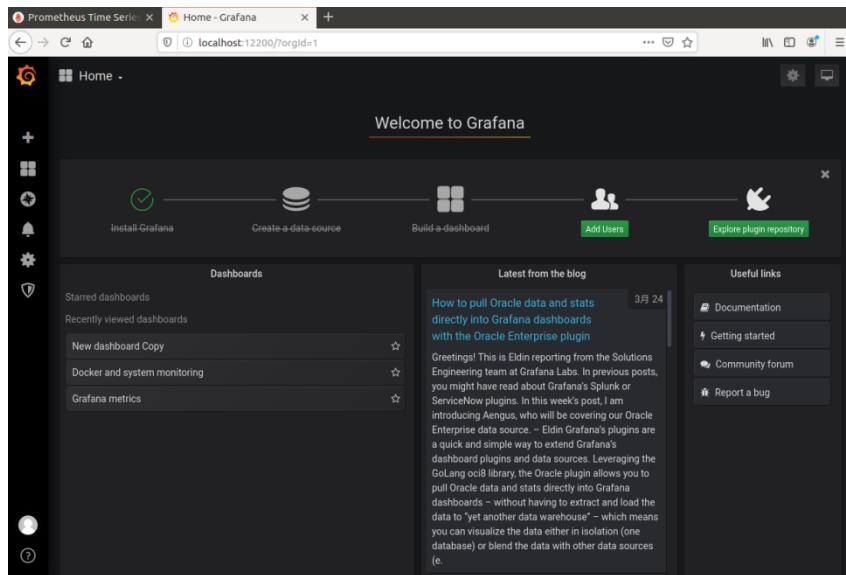


图 4.8 Grafana 界面

### 4.3 监控指标设计

由于对区块链内部工作原理及具体工作机制没有深入了解，进行监控的指标为整体性能指标，它涉及到区块链交易时的整体资源消耗以及吞吐量，而不深入到对区块链内部，对合约内某个行为进行测试。如下这些指标可以使管理者或用户对区块链性能有一个整体的了解。

**每秒事务数：**不同的区块链在部署、调用和执行智能合约上的速度不同。因此最重要和直观的是监控一段时间内区块链系统的吞吐量，即每秒的事务数。假设  $t_i$  到  $t_j$  时间内产生的事务数为  $Tx$ ，TPS 为 transactions per second 的缩写，如式(4-1)：

$$TPS = \frac{Tx}{t_j - t_i} \text{ (tx/s)} \quad \text{式(4-1)}$$

**事务平均响应延迟：**指区块中事务提交并确认有效的时间间隔。如果用户在区块链上部署智能合约，则必须等待一段时间，直到合约被确认，然后合约才能响应其他操作。也就是说，与集中式合约或软件相比，区块链上的智能合约以增加响应延迟为代价来提高可靠性。下式中  $Tx$  指一段时间内响应的事务总数，ARD 表示 average response duration，如式(4-2)：

$$ARD = \frac{\sum (T_{\text{确认}} - T_{\text{提交}})}{Tx} (\text{s}) \quad \text{式(4-2)}$$

每 CPU 周期完成的事务数：即衡量完成一个事务所消耗的 CPU 资源。在智能合约的执行过程中，它消耗了大量的 CPU 资源。CPU 消耗的程度由智能合约中实现的业务逻辑决定。与加密，循环有关的合约将消耗大量的 CPU 资源。提交块、计算块的散列的行为也消耗了大量的 CPU 资源。不同的区块链系统所在的 CPU 性能可能不同，因此需要一个度量来监视运行智能合约时 CPU 的利用率，本文使用“每 CPU 周期完成的事务数”来表示，如式(4-3)，其中 Tx 指时间段内完成的事务总数，f 为 CPU 频率，单位为 GHz，CPU(t) 则指在 t 时刻的 CPU 利用率。

$$TPC = \frac{Tx}{\int_{t_i}^{t_j} f * CPU(t)} (\text{tx/GHz} \cdot \text{s}) \quad \text{式(4-3)}$$

单位存储用量完成的事务数：在合同执行过程中，Docker 将从全局状态加载相关的帐户数据，并使用数组或其他操作，这些都会占用存储空间。该公式表现了区块链对存储资源的消耗程度，式(4-4)中 TPMS 指 Transactions per Memory\*Second，Memory(t) 表示 t 时刻占用的存储空间：

$$TPMS = \frac{Tx}{\int_{t_i}^{t_j} Memory(t)} (\text{tx/MB} \cdot \text{s}) \quad \text{式(4-4)}$$

单位 IO 用量完成的事务数：同内存用量消耗一样，在合约执行过程中也会消耗磁盘资源，用下述公式来描述区块链系统对磁盘资源的使用情况，式(4-5)中 TPIO 表示“每磁盘 IO 数据完成的交易量”(Transactions per Disk IO)，DISKR(t) 和 DISKW(t) 分别表示 t 时刻磁盘读写速度：

$$TPDIO = \frac{Tx}{\int_{t_i}^{t_j} DISKR(t) + DISKW(t)} (\text{tx/KB}) \quad \text{式(4-5)}$$

单位网络传输量完成的事务数：在区块链系统中，对等节点的通信需要消耗网络流量，如使用共识机制来使节点达成一致，在网络中传输事务或是数据同样消耗网络资源。计算公式见式(4-6)，其中 TPN 为 Transactions per Network Data 的缩写，UPLOAD(t) 和 DOWNLOAD(t) 表示 t 时刻的上传速率和下载速率：

$$TPN = \frac{Tx}{\int_{t_i}^{t_j} UPLOAD(t) + DOWNLOAD(t)} (\text{tx/KB}) \quad \text{式(4-6)}$$

#### 4. 4 详细监控设计

在上文配置好所需环境后，本章节将根据三种主流区块链系统：Fabric1.4，Ethereum，IOTA 分别介绍详细的指标获取方式以及目的指标的计算方法，以完整

说明测试流程。

#### 4.4.1 Fabric1.4 监控设计

Fabric 的节点与比特币、以太坊的不同之处，主要是其节点会扮演不同的角色，如背书、提交、排序等，每个角色有各自的任务，在测试的过程中部署了十台服务器节点，节点各自角色如下，如图 4.9，其中 peer0 与排序节点部署于同一台机器。

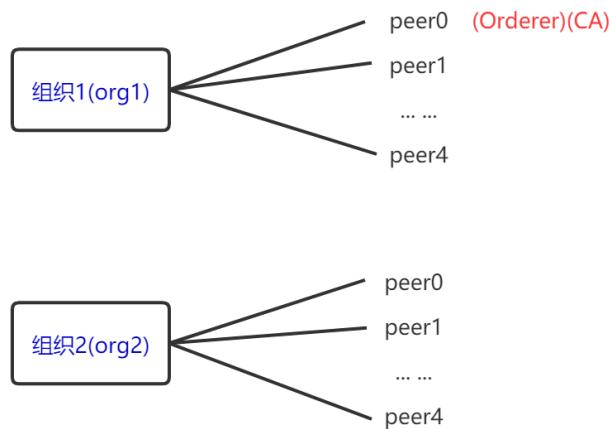


图 4.9 Fabric 节点部署结构

Fabric 交易流程如图 4.10 所示，客户端在通过认证后，首先利用 SDK 构造交易提案并发送给背书节点，节点验证交易的签名并模拟执行交易，并打上自己的签名，随后将结果返回给客户端，客户端判断各背书节点结果一致后，便可提交交易给排序节点，排序节点按照设定的共识机制进行排序并生成区块，最后提交至提交节点进行校验，完成后将区块追加至本地区块链上。

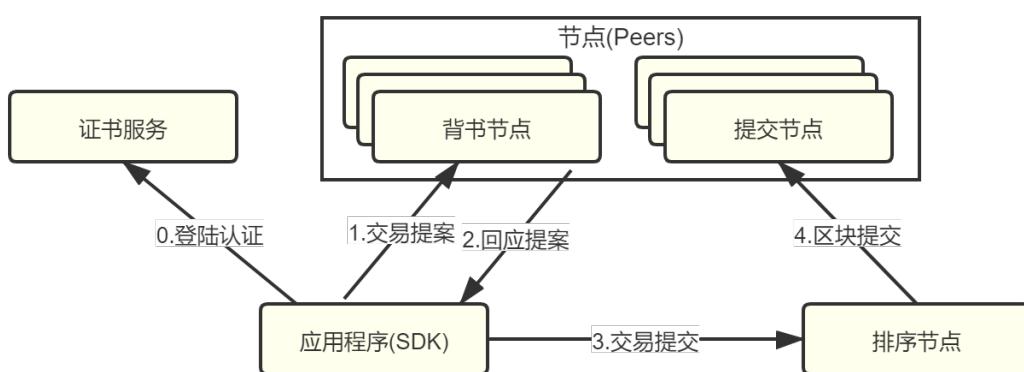


图 4.10 Fabric 交易流程

在测试中，排序节点使用 Solo 共识机制来进行排序，即只使用一个排序节点

来排序和产生区块，因此所有交易都会经过这个排序节点，统计排序节点收集到的交易数据即可反映区块链的交易量。

### 1) 容器数据收集：

从 cadvisor 收集到的区块链系统占用的容器资源指标如表 4.6：

表 4.6 Fabric1.4 的系统资源使用指标

指标内容		指标名称
cpu 使用量		container_cpu_usage_seconds_total{name="orderer.example.com"}
磁盘 用量	读	container_fs_reads_bytes_total{name="orderer.example.com"}
	写	container_fs_reads_bytes_total{name="orderer.example.com"}
网络 用量	上传	container_network_transmit_bytes_total{name="orderer.example.com"}
	下载	container_network_receive_bytes_total{name="orderer.example.com"}
内存用量		container_memory_usage_bytes{name="orderer.example.com"}

### 2) 区块链数据收集：

Fabric1.4 自身对外提供可由 Prometheus 收集的时序数据，开放端口为 8443，下图 4.11 为数据库可获取到的部分指标截图。

[Docs](#) » [Operations Guides](#) » [Metrics Reference](#) [Edit on GitHub](#)

## Metrics Reference

### Orderer Metrics

#### Prometheus

The following orderer metrics are exported for consumption by Prometheus.

Name	Type	Description
blockcutter_block_fill_duration	histogram	The time from first transaction enqueue to block creation.
broadcast_enqueue_duration	histogram	The time to enqueue a transaction into the broadcast channel.
broadcast_processed_count	counter	The number of transactions processed by the orderer.
broadcast_validate_duration	histogram	The time to validate a transaction in the orderer.

图 4.11 Fabric1.4 排序节点的指标

其中区块链的交易量指标为：broadcast\_processed\_count，确认交易有效的时延为 broadcast\_validate\_duration，通过前者可以计算出 TPS，后者用于计算 ARD。

### 3) 目的指标：

在本次设计中，部分指标需要通过 PromQL 运算得到，Fabric1.4 的运算公式设计如下表 4.7 所示，其统计间隔为 5 秒，为了方便表示，使用 MSE 代替

{channel="mychannel", status="SUCCESS", type="ENDORSER\_TRANSACTION"},  
使用 OEC 代替 {name = "orderer.example.com"}。

表 4.7 Fabric1.4 目的指标查询公式

TPS	increase(broadcast_processed_count{MSE}[5s])
TPC	$\frac{\text{sum}(\text{increase}(\text{broadcast\_processed\_count}\{\text{MSE}\}[5\text{s}]))}{\text{sum}(\text{increase}(\text{container\_cpu\_usage\_seconds\_total}\{\text{OEC}\}[5\text{s}])) * 2.5}$
TPMS	$\frac{\text{sum}(\text{increase}(\text{broadcast\_processed\_count}\{\text{MSE}\}[5\text{s}])) * 1024 * 1024 / 5}{\text{sum}(\text{increase}(\text{container\_memory\_usage\_bytes}\{\text{OEC}\}[5\text{s}])) / 2 + \text{sum}(\text{container\_memory\_usage\_bytes}\{\text{OEC}\})}$
TPIO	$\frac{\text{sum}(\text{increase}(\text{broadcast\_processed\_count}\{\text{MSE}\}[5\text{s}])) / 1024}{\text{sum}(\text{increase}(\text{container\_fs\_reads\_bytes\_total}\{\text{OEC}\}[5\text{s}])) + \text{sum}(\text{increase}(\text{container\_fs\_writes\_bytes\_total}\{\text{OEC}\}[5\text{s}]))}$
TPN	$\frac{\text{sum}(\text{increase}(\text{broadcast\_process\_count}\{\text{MSE}\}[5\text{s}])) / 1024}{\text{sum}(\text{increase}(\text{container\_network\_receive\_bytes\_total}\{\text{OEC}\}[5\text{s}])) + \text{increase}(\text{container\_network\_transmit\_bytes\_total}\{\text{OEC}\}[5\text{s}]))}$

#### 4.4.2 Ethereum 监控设计

以太坊<sup>[27]</sup>节点分为全节点和轻节点，两者只在数据存储上有所区别，前者保存有全量交易信息，而后者只保存有区块的区块头，用户可以根据自己的设备容量、性能等选择节点。本文搭建了拥有十个节点的以太坊网络进行测试。在启动时需要完成 30303 以及 8545 的端口映射，前者提供了以太坊客户端节点互连的接口，以完成网络搭建，后者则是 RPC 接口<sup>[28]</sup>，以发送交易、查询交易量和时延。

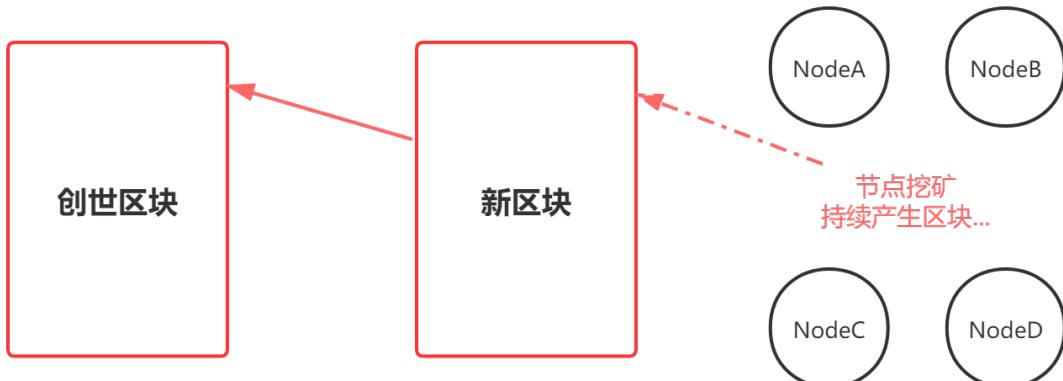


图 4.11 Ethereum 测试网络

### 1) 容器数据收集:

从 cadvisor 收集到的区块链系统占用的容器资源指标与 Fabric1.4 的指标相比，区别只在于标签不同，此时数据查询使用的标签为 `images="ethereum/client-go:v3.0"`，指标名称是相同的，因此在此处不再介绍，见表 4.6 即可。

### 2) 区块链数据收集:

由于 Ethereum 不像 Fabric1.4 一样，提供可由时序数据库读取的 metrics，因此需要通过远程过程调用(RPC)来获取其吞吐量和时延，在本文中使用 python 实现 RPC 请求，并将获取的数据进行处理得到吞吐量和时延，并转换成 metric，最后打开特定端口供数据库访问<sup>[29]</sup>。

获取吞吐量的算法分为三个步骤：计算一段  $t$  时间内产生的区块，编号从 `block(i)` 直到 `block(n)` 计算其中新产生的区块内打包的交易量之和  $Tx = \sum_{k=i+1}^n block(k) * transactions(k)$  TPS=Tx/t，输出吞吐量。算法见下表 4.8：

表 4.8 Ethereum 吞吐量计算方法

```
#查询当前区块数为 n
n=requests.post('http://localhost:8545',data=json.dumps(eth_blockNumber),headers=
                 headers)
#等待 t 时间后再次查询区块数为 m,得到这段时间内产生了 m-n 个区块
time.sleep(t)
m=requests.post('http://localhost:8545',data=json.dumps(eth_blockNumber),headers=
                 headers)
blocks=m-n
#查询每个产生的区块中的交易量并求和
for i in range from block(n) to block(m):
    transactions=requests.post('http://localhost:8545',data=json.dumps(getBlockTransa
                               tionCountByNumber), headers=headers)
    Tx+=transactions
#计算吞吐量
TPS=Tx/t
#创建指标，类型为 gauge，并通过 http server 启动 12580 端口输出数据
res=Gauge('TPS_Ethereum', 'Throughput of Ethereum')
http_server_start(10001,'10.0.2.15')
while(true):
    res.set(TPS)
```

以太坊时延(ARD)的计算与吞吐量类似，也通过 RPC 访问区块链，来得到交易数据：发送交易在区块链上查询此交易计算时延。实际测试时多次调用该算法，

得到交易时延的平均值。算法如下表 4.9:

表 4.9 Ethereum 时延计算方法

```
#发送交易并记录时间
hash=requests.post('http://localhost:8545',data=json.dumps(transaction),headers=he
ders)
time1=time.time()
#查询直到该交易被添加到区块链位置，未确认的交易返回值是空
res=None
while(res is None):
    res=requests.post('http://localhost:8545',data=json.dumps(TransByhash),headers=h
eaders)
#计算时延
ARD_Ethereum=time.time()-time1
```

将时延数据转换为 metrics 的方法与吞吐量类似，此时启动的是 12345 端口。运行 python 程序以后，可以从网页分别打开两个端口，查看到 TPS\_Ethereum 与 ARD\_Ethereum 两项指标数据。

### 3) 目的指标:

在基准测试和 Grafana 可视化中，使用表 4.7 的计算公式形式即可，其中吞吐量和延迟分别为 TPS\_Ethereum 和 ARD\_Ethereum，资源占用的指标则将分子改为以太坊的吞吐量即可，例如式(4-7):

$$TPC = \frac{\text{sum}(TPS\_Ethereum * 5)}{\text{sum}(\text{increase}(\text{container\_cpu\_usage\_seconds\_total}\{\text{id}\}[5s])) * 2.5} \quad \text{式(4-7)}$$

### 4.4.3 IOTA 监控设计

在 IOTA<sup>[30]</sup>的分布式 P2P 网络中，存在节点和协调器两种实体。节点包括全节点和轻节点，前者为同步所有区块链数据的节点，后者则只包含区块头。协调器(Coordinator)则负责发布里程碑(用来使协调器确认的交易达到 100% 确认置信度)，由于 IOTA 是为了应用规模而建立，而当前交易量不足以避免双花攻击，因此需要协调器来处理安全问题，在未来，当 IOTA 演变成熟，完整的 Tangle 分布式共识算法开始发挥作用，IOTA 基金会将关闭协调器，使网络更有效率。

本文在部署 IOTA 时采用图 4.12 的方式，节点组成 P2P 通信网络，这称为 IRI (IOTA Reference Implementation)，即一个 IOTA 系统实例，并部署了一个协调器，随后进行测试。

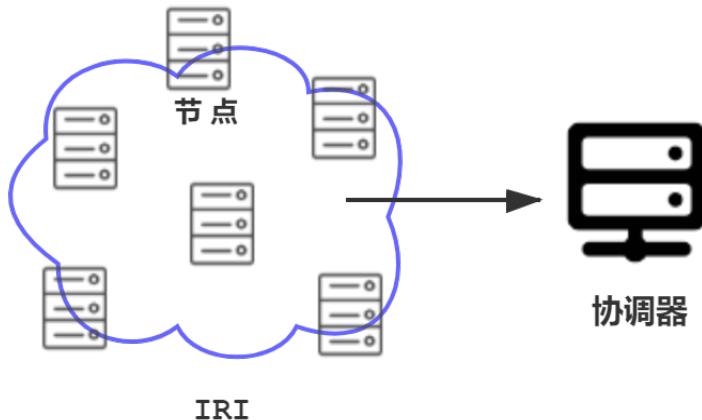


图 4.12 IOTA 部署结构图

### 1) 容器数据收集:

在 IOTA 测试中，也需要使用 cadvisor 收集容器数据，其指标与 Fabric1.4 的指标类似，见表 4.6，其中容器指标名称相同，标签中镜像名称不同，为 `image=""`。

### 2) 区块链数据收集:

IOTA 运行时，其节点通过自身服务，会收集区块链实时数据并规范化，并从端口 5556 输出，IOTA-exporter 从 5556 读取数据后，将数据整理成 metrics，并从 9311 端口输出，使得数据库能够读取相应指标。区块链提供了 `iota_zmp_seen_tx_count` 作为交易总量，因此其一分钟内的平均吞吐量可用 `sum(rate(iota_zmp_seen_tx_count[1m]))` 来计算。同时，端口提供了 `iota_zmp_tx_confirm_time_sum` 表示所有交易的总确认时延，以及 `iota_zmp_tx_confirm_time_count` 总确认交易数，将前者处理为一段时间内交易时延的增量后，除以后者，就可以算出这段时间内的每个交易的平均时延。

### 3) 目的指标:

IOTA 的目的指标计算公式也可见表 4.7，其形式相同，分子改为 IOTA 的吞吐量和时延即可。

## 4.5 本章小结

本章具体讲解了测试系统使用到的各个应用及其配置方法，对系统框架中的 Cadvisors、Prometheus、Grafana 等进行了详细的讲解，包括如何使用、如何配置，以及一些问题的解决方案，并对各区块链的具体测试指标做了介绍。下一章将给出区块链测试结果，并展示区块链系统的实时监控图。

## 第五章 测试与监控

### 5.1 基准测试结果

1) 对于吞吐量和时延的测试结果如下:

表 5.1 三种区块链项目整体性能

项目名称	吞吐量(tx/s)	时延(s)
Ethereum	150	100
IOTA	250	39
Fabric1.4	765	5

从表 5.1 中可以看出 Ethereum 的吞吐量在三者中最低并且时延最高, IOTA 中规中矩, Fabric 的整体性能则最好。Ethereum 和 IOTA 都采用 PoW 的共识机制, 虽然它保证了完全的去中心化和安全性, 但是其低吞吐量和高时延都使得在工作负载较高的场景中部署 IOTA 和 Ethereum 可能无法满足应用需求, 未来, Ethereum 转向权益证明机制后, 其性能会提升, 资源消耗也会降低。而 Fabric 并不是完全去中心化的项目, 因此它能处理的交易量更大, 处理时延也更低。

2) 资源利用率测试结果:

表 5.2 三种区块链资源利用率

项目名称	TPS	TPC(tx/GHz*s)	TPMS(tx/MB*s)	TPDIO(tx/KB)	TPN(tx/KB)
Ethereum	11	2.8	0.006	0.19	0.26
IOTA	2.5	2.8	0.0008	0.02	0.036
Fabric1.4	93	414	0.1	0.082	0.19

从上表 5.2 中的数据可以看出以太坊对于磁盘 IO 和网络资源的利用率比较高, 对于内存和 CPU 的利用率较低, 可能是 PoW 机制在 hash 计算上消耗了大量资源, 因此在高频交易的场景中应尽量避免使用 PoW 区块链系统, 而 IOTA 各项指标都处于低水平, 因此虽然它采用 Tangle 作为数据结构, 并且交易无手续费, 天然适合物联网场景, 但从资源角度分析, 它的低资源利用率表明它在面对资源严重受限的物联网节点时力不从心, 需要对区块链进行更多优化, 在本次测试中, 其资源利用率及吞吐量的低水平, 还有可能是部署的节点较少, 没有充分发挥其架构优势。Fabric 对 CPU 和内存的利用率相较于前两者来说很高, 是因为本文在测试中采用 solo 的共识机制, 只有一个排序节点在处理交易顺序, 结构简单, 对计算机 CPU 的消耗较少, 但其 TPN 数值较低, 如果网络规模增大, 节点通信的成本增加, 那它在网络资源利用的劣势将会使系统性能急剧降低, 因此它目前还是比较适合部

署在规模较小的应用中。

## 5.2 实时监控结果

1) Fabric1.4:

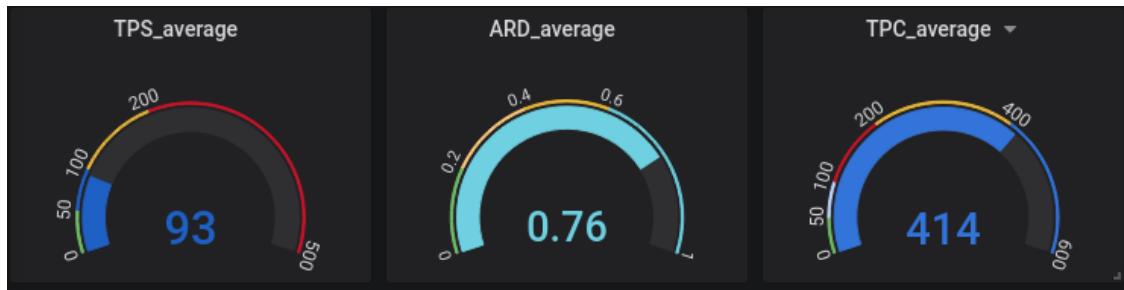


图 5.1 Fabric1.4 吞吐量、时延、TPC 的平均值

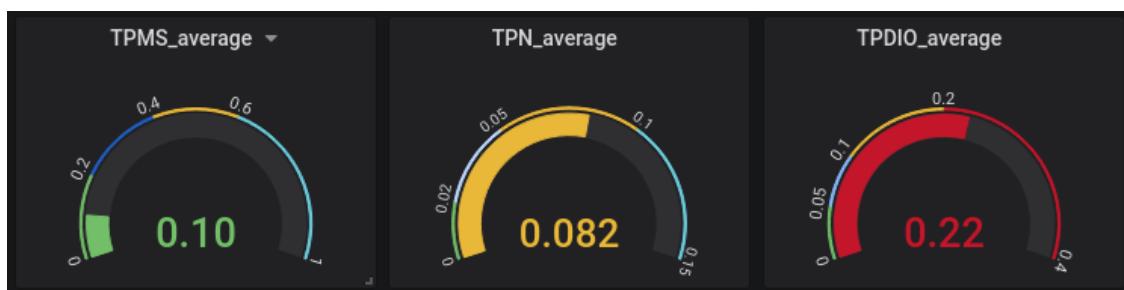


图 5.2 Fabric1.4 中 TPMS、TPN、TPDIO 的平均值

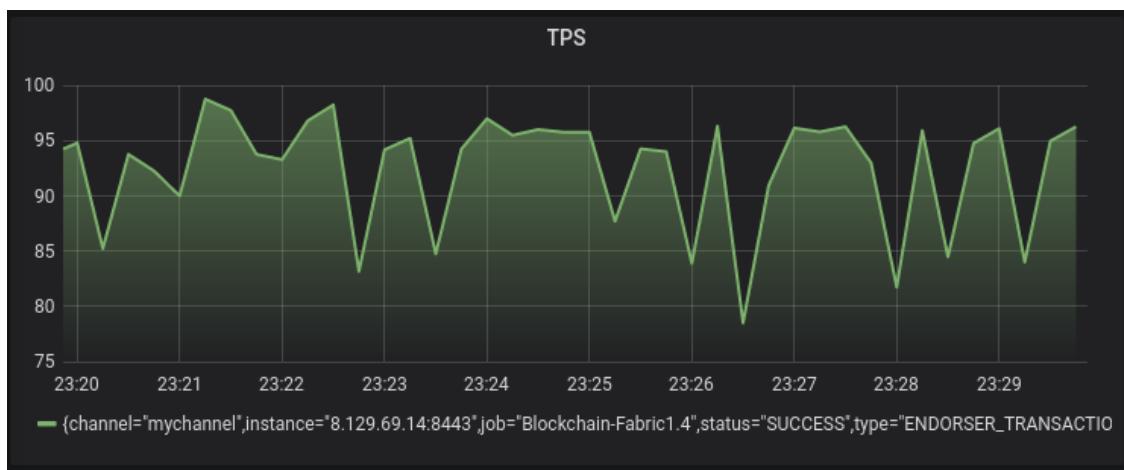


图 5.3 Fabric1.4 实时吞吐量监控

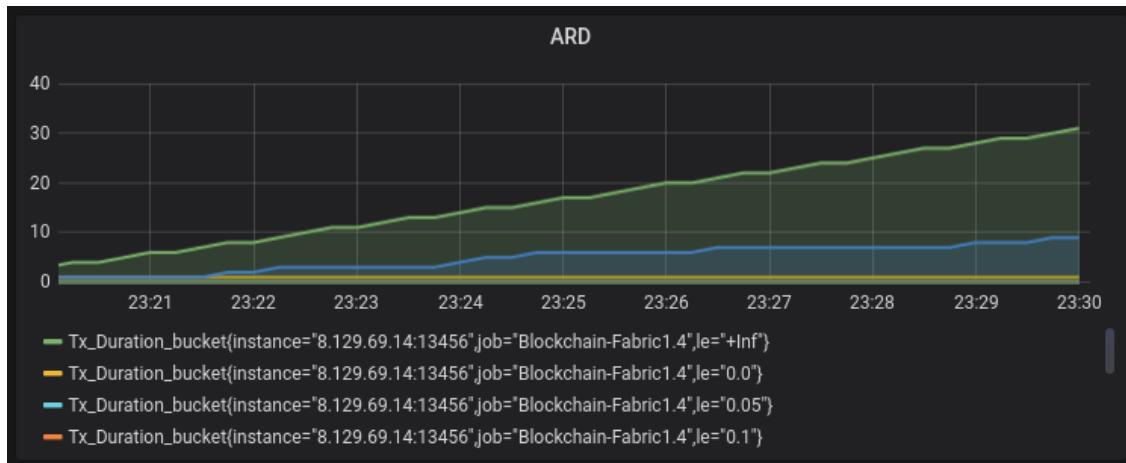


图 5.4 Fabric1.4 交易时延统计

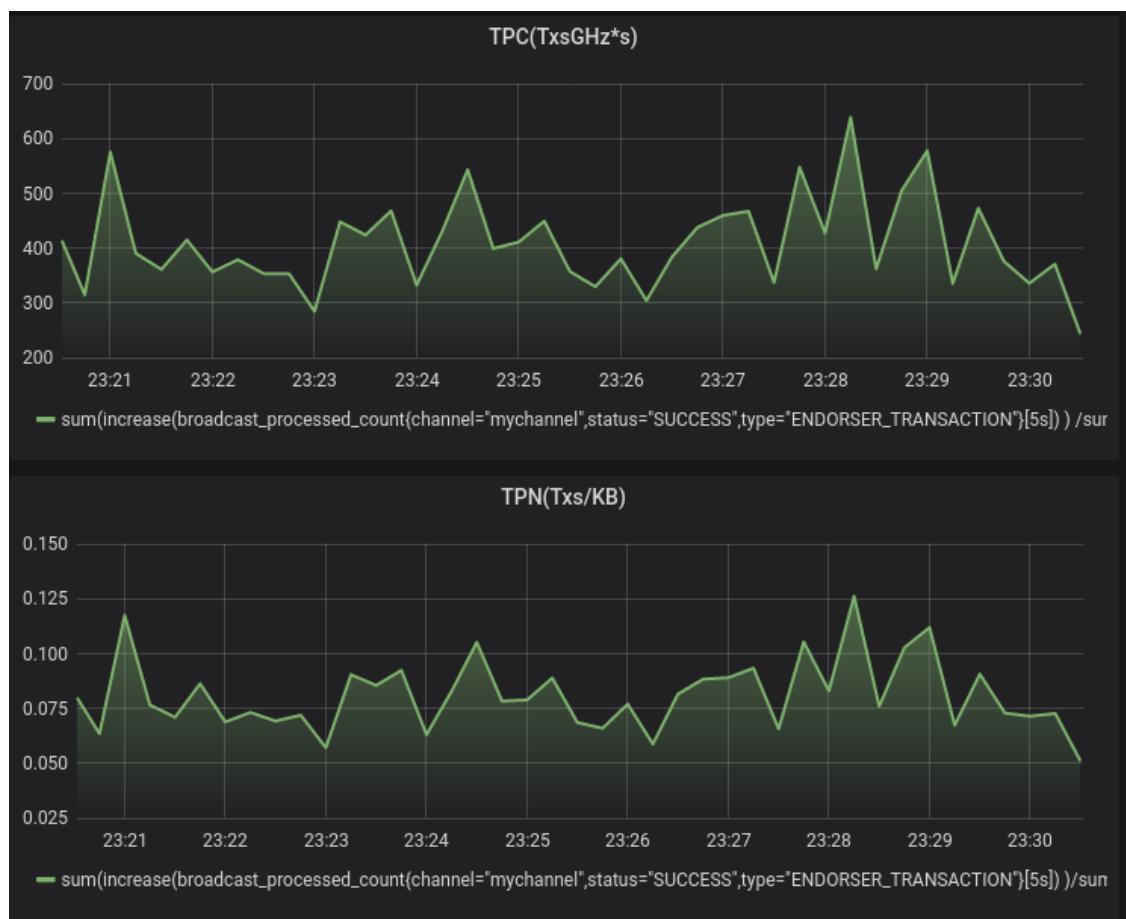


图 5.5 Fabric1.4 中 TPC (上) 与 TPN (下) 监控

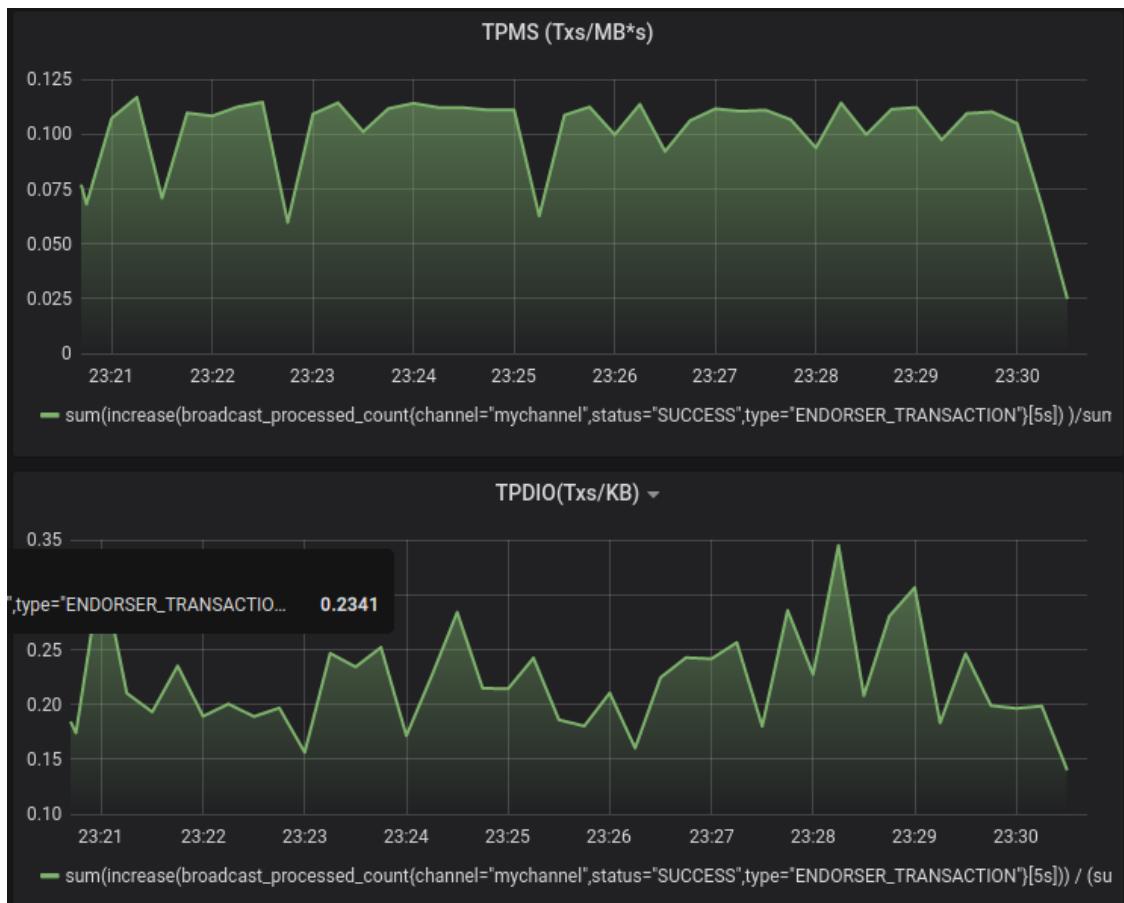


图 5.6 Fabric1.4 中 TPMS（上）与 TPDIO（下）监控

## 2) IOTA:

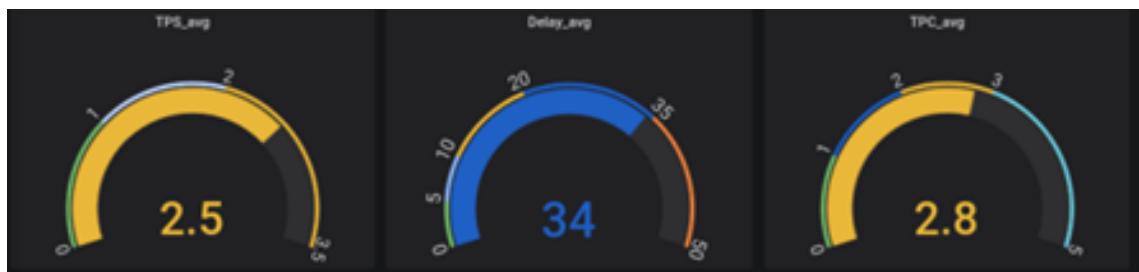


图 5.7 IOTA 吞吐量、时延、TPC 的平均值

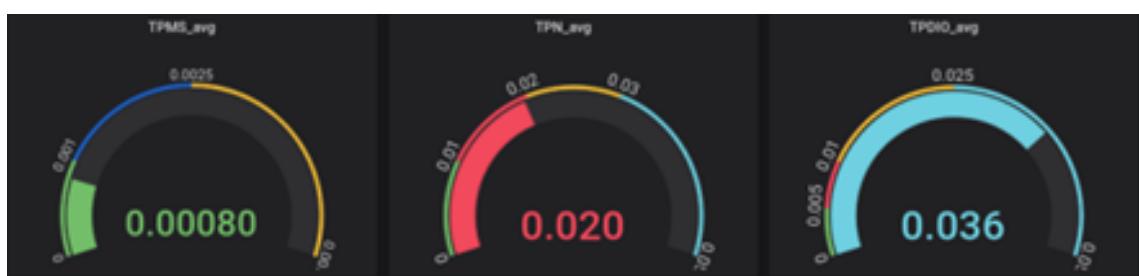


图 5.8 IOTA 中 TPMS、TPN、TPDIO 的平均值

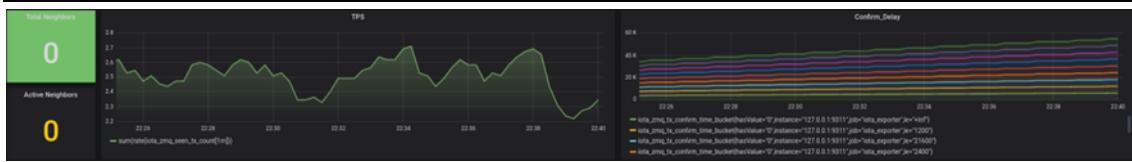


图 5.9 IOTA 中邻居节点个数、吞吐量、时延监控

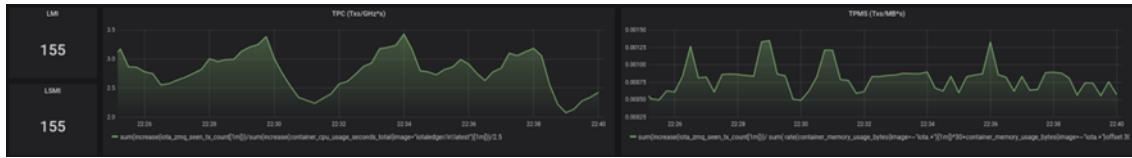


图 5.10 IOTA 中里程碑数量、TPC、TPMS 监控

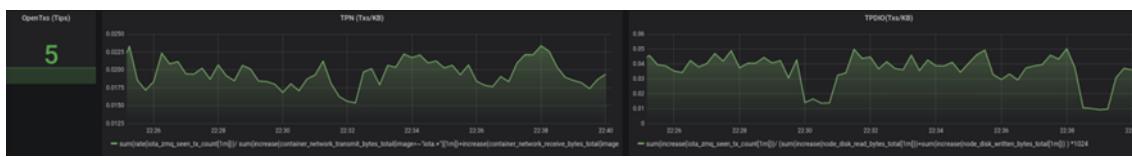


图 5.11 IOTA 中 Tips 数量、TPND、TPDIO 监控

### 3) Ethereum:



图 5.12 Ethereum 中平均吞吐量、时延、TPMS 以及吞吐量监控

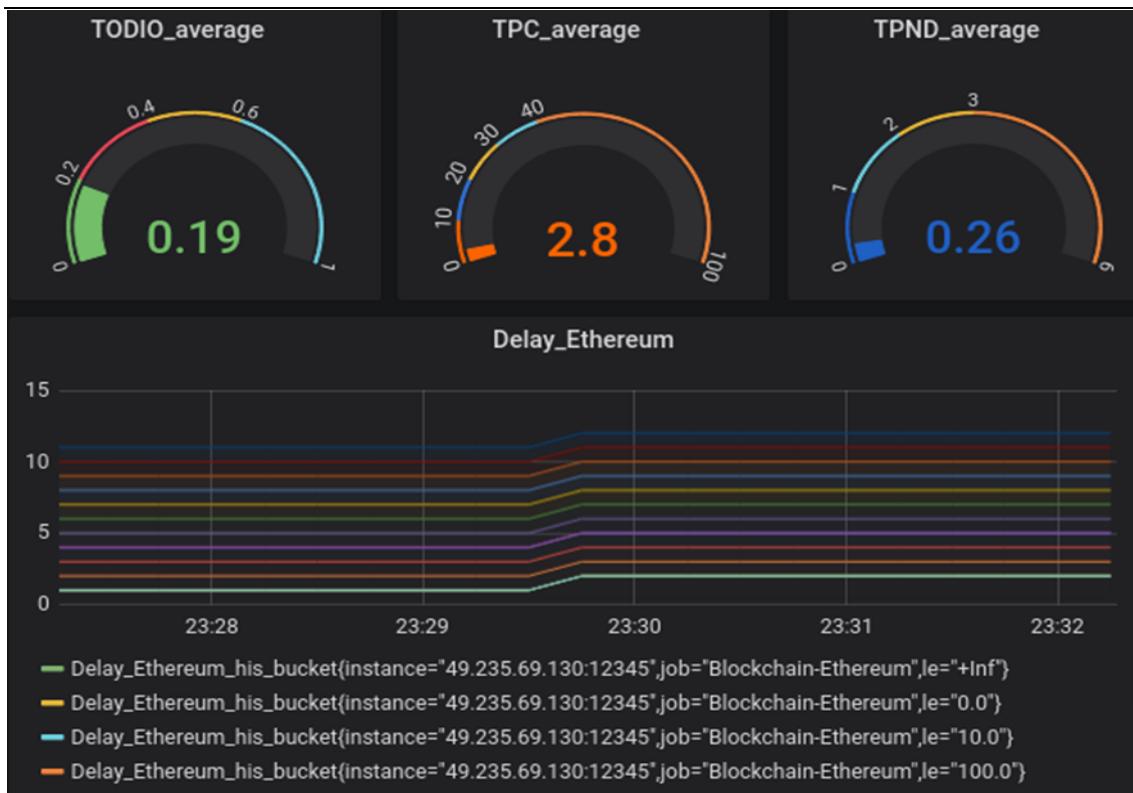


图 5.13 Ethereum 中平均 TPDIO、TPC、TPND 以及时延监控

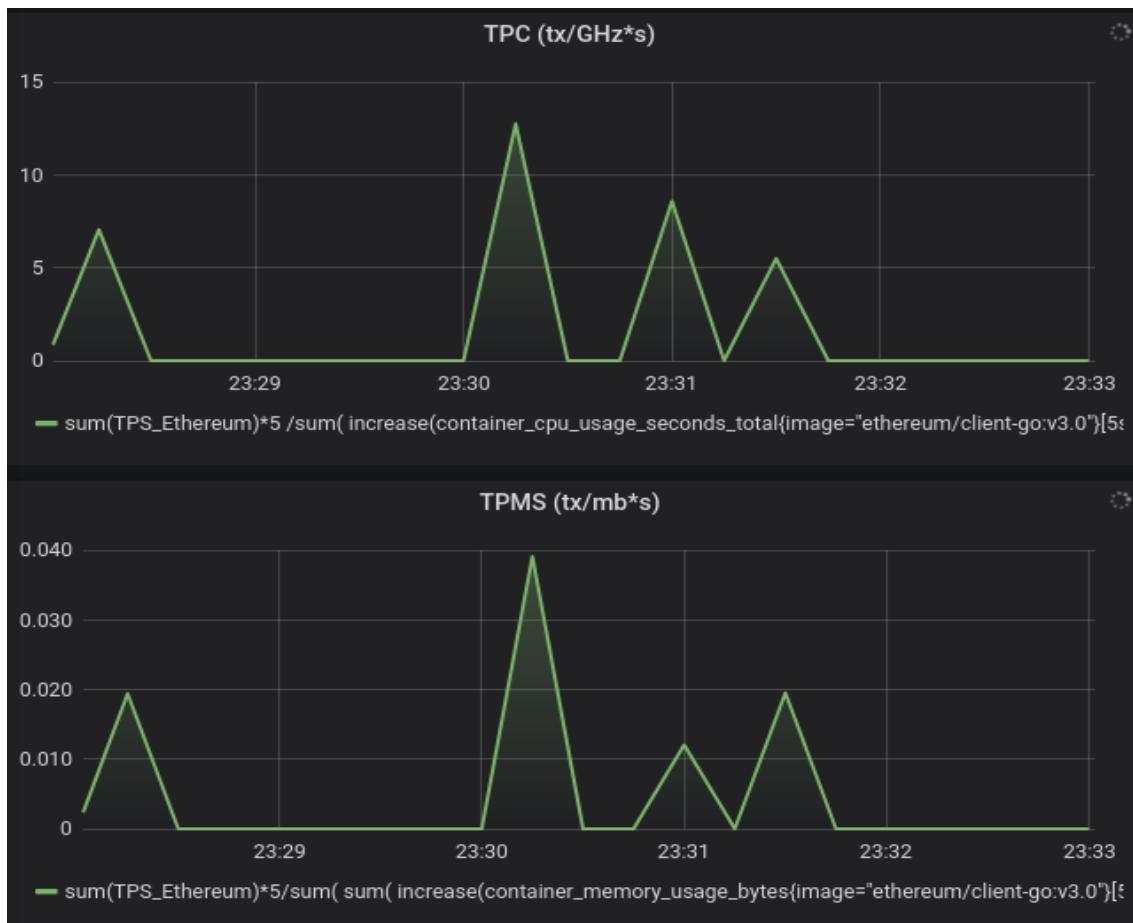


图 5.14 Ethereum 中 TPC 及 TPMS 监控

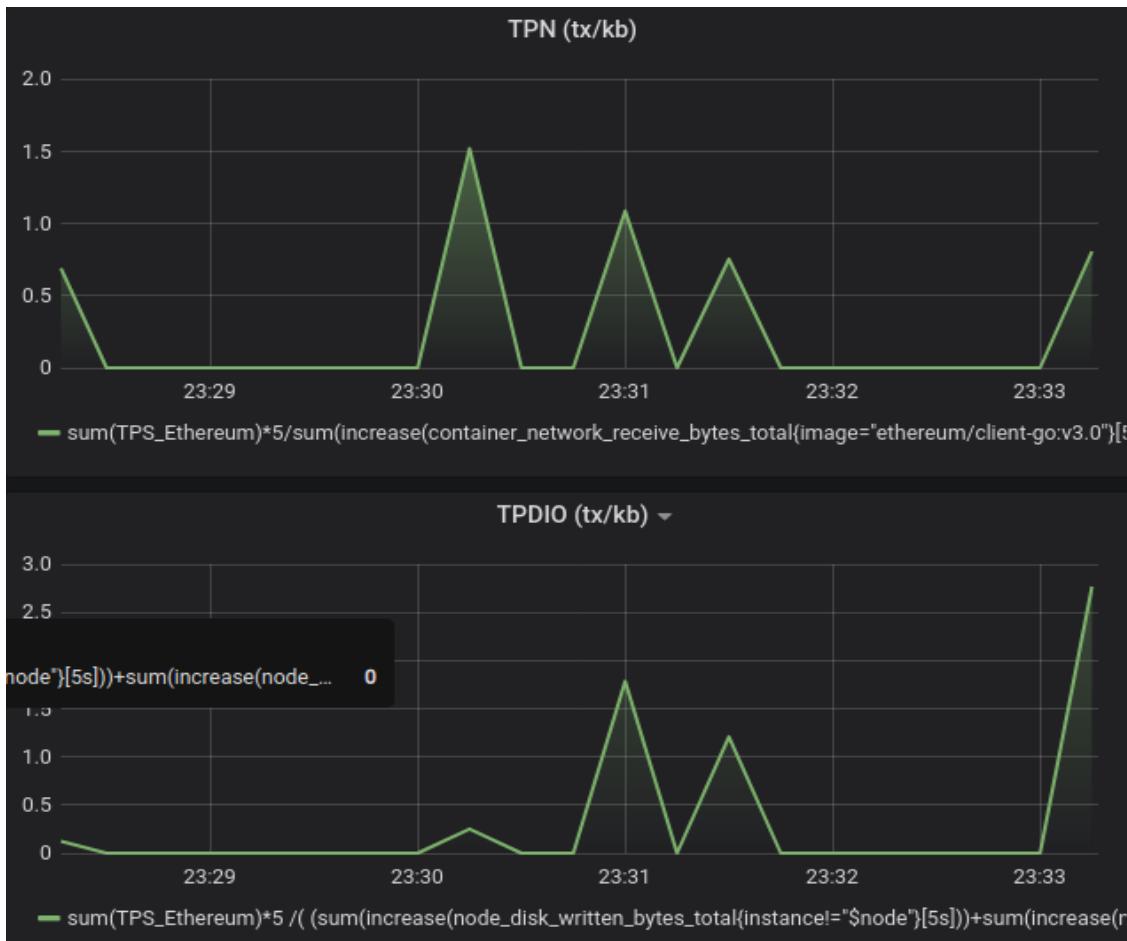


图 5.15 Ethereum 中 TPN 及 TPDIo 监控

#### 4) 性能分析:

在一般交易量下，对三种主流区块链进行测量后得到图 5.1 至图 5.15 的监控图，由于 Prometheus 在数据抓取过程中可能出现遗漏，因此监控图形在部分样本点上可能出现失真，但大体上正确。同时，本文在测试时将以太坊挖矿难度调整较大，因此其 TPS 较低，对于 IOTA，则是节点数目较少，没有发挥其 Tangle 结构的优势，因此其性能很差，Fabric1.4 的测试则完全体现了它应用多中心化特性所带来的的吞吐量和性能优势。

## 结束语

通过本次毕业设计，不仅让我对区块链这个蓬勃发展的技术有了更多了解，也使我对许多新型应用如 Prometheus 和 Grafana 有了深刻的认识，让我对运维和开发有了新的理解，更重要的是，本次毕设让我初步学习了如何阅读应用的开发文档，如何筛查所需要的论文资料，进一步提升了我的学习能力。

通过测试和监控我们可以发现当今区块链系统的性能还不能满足大规模应用的需要，在面对高并发、大流量的任务时有些力不从心，但各个区块链系统都在进行积极的改进。在众多学者和技术人员的不懈努力下，相信在不久的将来，我们会看到区块链与物联网、云计算等技术成功结合并出现在实际应用中。

在设计中所使用到的都是近年受到开发和运维人员喜爱的开源应用，在 GitHub 上都开源了各自的代码，阅读源码是研究应用和提高开发能力的重要方法。本文对 Prometheus 的使用只是初步的，它还有更多定制方案，通过开发能够对当今众多软件进行监控和测试，这需要对 python 和 go 等编程语言有更深的理解。另外，在设计中对 Grafana 的图形应用也不够美观，有很多的个性化设置没有使用到，本测试系统还有发展完善的空间。

## 致 谢

写到这里，不仅仅说明毕业论文的编写工作将要告一段落，也表明四年大学生活即将结束。回顾在南京邮电大学学习的四年，从入学的通信展览馆参观，到各种软硬件实验课，我学习到了不少理论和实践的知识，再到最后的毕业设计，通过大四下学期的时光，使我对前沿技术以及应用软件有了新的了解。时间飞逝，新生入学报到，坐满讲堂的景象仿佛就在昨日，但明天我们就要踏入社会了。在四年的学习生活中，我要感谢南邮各位老师的认真教学、辅导员的细心照顾以及同学们长久的陪伴。

我感谢要本次毕业设计的指导老师王堃，感谢对我设计的指导以及对论文的严格审查，正是老师的教导才使我的毕设能达到要求，更重要的是使我学到了能实际应用的知识。王老师在学术上严谨的态度，对时间的严格安排，和对我们耐心的指导是此次毕设完成的关键。我还要感谢舍友刘寅秋对我的帮助，他在系统设计和实验测试上提供了许多资料和参考意见，对于这样一个综合的系统，不仅涉及到区块链部署，还有 python 编程，也涉及到其他我以前未曾学习过的知识，若没有他的帮助，完成这样一个课题对我来说是一件非常困难的事情。我也要感谢韩亚敏学姐的认真和负责，用自己的时间仔细查阅我各个文档并督促我修改格式，发现并指出我的各种错误，使毕业设计能按进度完成。管星学姐则不厌其烦地帮我搭建和调试区块链系统，为完成以太坊测试提供了宝贵的经验。

最后，我要感谢学校老师抽出宝贵的时间对论文进行审查和评阅，并感谢参加论文答辩的教师们对我提出的宝贵意见和建议。

## 参考文献

- [1] 袁勇, 王飞跃. 区块链技术发展现状与展望[J]. 自动化学报, 2016, 42(4):481-494.
- [2] 区块链(数据结构)\_百度百科 [EB/OL].[2020-01-01].<https://baike.baidu.com/item/区块链/13465666>.
- [3] 郑敏, 王虹, 刘洪, 谭冲. 区块链共识算法研究综述[J]. 信息网络安全, 2019, 19(7):8-24.
- [4] Christidis K, Devetsikiotis M. Blockchains and Smart Contracts for the Internet of Things [J]. IEEE Access, 2016, 4:2292-2303.
- [5] 赵冠臣, 王冬妮, 刘至洋, 孟振江. 浅谈 Docker 容器技术[J]. 有线电视技术, 2019, 26(9): 85-88.
- [6] 何宇锋, 林奕琳, 张琳峰. 容器技术在移动核心网的应用[J]. 移动通信, 2018, 42(3): 27-32.
- [7] Boettiger C. An Introduction to Docker for Reproducible Research [J]. ACM SIGOPS Operating Systems Review, 2015, 49(1):71-79.
- [8] 王旭, 甘国华, 吴凌云. 区块链性能的量化分析研究[J]. 计算机工程与应用, 2020, 56(3): 55-60.
- [9] Nathan S, Thakkar P, Vishwanathan B. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform [C]// 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2018.
- [10] Dinh T T A, Liu R, Zhang M, et al. Untangling Blockchain: A Data Processing View of Blockchain Systems [J]. IEEE Transactions on Knowledge & Data Engineering, 2017, PP(99):1-1.
- [11] Idweesh A, Alharby M, Moorsel A V. Performance Benchmarking for Ethereum Opcodes [C]// 15th International Conference on Computer Systems and Applications (AICCSA). IEEE/ACS, 2018.
- [12] Moorsel A. Benchmarks and Models for Blockchain [C]// International Conference on Performance Engineering (ICPE). ACM/SPEC, 2018.
- [13] Ampel B, Patton M, Chen H. Performance Modeling of Hyperledger Sawtooth Blockchain [C]// International Conference on Intelligence and Security Informatics (ISI). IEEE, 2019.
- [14] Zheng P, Zheng Z, Luo X, et al. A Detailed and Real-time Performance Monitoring Framework for Blockchain Systems [C]// International Conference on Software Engineering Software Engineering in Practice (ICSE-SEIP). IEEE, 2018.
- [15] Gervais A, Karame G O, Karl W, et al. On the Security and Performance of Proof of Work Blockchains [C]// Special Interest Group on Security, Audit and Control (SIGSAC). ACM, 2016.
- [16] Dinh T T A, Wang J, Chen G, et al. BLOCKBENCH: A Framework for Analyzing Private Blockchains [C]// International Conference on Management of Data. ACM SIGMOD, 2017.
- [17] Xuchen D, Haochen P, Lewis T, et al. BBB: Make Benchmarking Blockchains Configurable and Extensible [C]// 24th Pacific Rim International Symposium on Dependable Computing (PRDC). IEEE, 2019.
- [18] Fabric 性能测试工具之 Caliper\_简书[EB/OL].[2018-11-25].<https://www.jianshu.com/p/1644810cf790>.
- [19] 田晓芸. 区块链技术驱动下的物联网安全研究综述[J]. 信息通信, 2019, (12):41-42.
- [20] 吴睿, 陈金鹰, 邓洪权. 区块链在云计算技术领域的应用[J]. 现代传输, 2019, (5):68-70.
- [21] Ruan P, Chen G, Dinh T T A, et al. Fine-grained, Secure and Efficient Data Provenance on Blockchain Systems [C]// Very Large Data Bases (VLDB). ACM, 2019.
- [22] 蒋润祥, 魏长江. 区块链的应用进展与价值探讨[J]. 甘肃金融, 2016, 455(2):20-22.

- [23] Elli A, Barger, Bortnikov A, Bortnikov V, et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains [C]// the Thirteenth EuroSys Conference. ACM, 2018.
- [24] Dhakate S and Godbole A. Distributed Cloud Monitoring Using Docker as Next Generation Container Virtualization Technology [C]// 2015 Annual IEEE India Conference (INDICON), IEEE, 2015.
- [25] Prometheus [EB/OL]. [2020-01-28]. <https://github.com/prometheus/prometheus>.
- [26] Gauge panel [EB/OL]. [2020-01-20]. <https://grafana.com/docs/grafana/latest/features/panels/gauge/>.
- [27] Chen T, Zhu Y, Li Z, et al. Understanding Ethereum via Graph Analysis [C]// International Conference on Computer Communications (INFOCOM). IEEE, 2018.
- [28] Ethereum/wiki/JSON-RPC [EB/OL]. [2020-02-03]. <https://github.com/ethereum/wiki/wiki/JSON-RPC#go>.
- [29] Client\_Python [EB/OL]. [2020-03-06]. [https://github.com/prometheus/client\\_python](https://github.com/prometheus/client_python).
- [30] PyOTA [EB/OL]. [2020-03-19]. <https://github.com/iotaledger/iota.py>.

## 附录 A 系统使用说明书

### 1. 编写目的

本文档为南京邮电大学 2016 级 2020 届本科生毕业设计，《基于区块链的基准测试系统》的使用说明书，旨在对初次使用此系统的人员进行指导。说明书以 Ethereum 为示例，展示了从应用的安装到详细配置，到区块链测试的全部过程，IOTA 和 Fabric1.4 则不再对测试过程进行介绍，方法与以太坊类似。通过该文档读者可以了解该系统的所有功能以及可选用的环境配置方法和使用方法。

### 2. 系统概要

本系统旨在为不同的区块链项目提供一个可移植性较好、测量较准确的通用基准测试框架，不仅可以实现基准测试，也可以对区块链进行实时监控，观测其性能变化。该系统由数据收集、数据存储、数据分析三个模块组成。数据收集由 cadvisor 以及 prometheus\_client 完成，数据持久化由时序数据库 Prometheus 完成，数据分析和可视化则由 Grafana 实现，这几种应用都是当下非常活跃的开源应用，拥有广阔的运维和开发前景，系统结构如图 A1 所示。

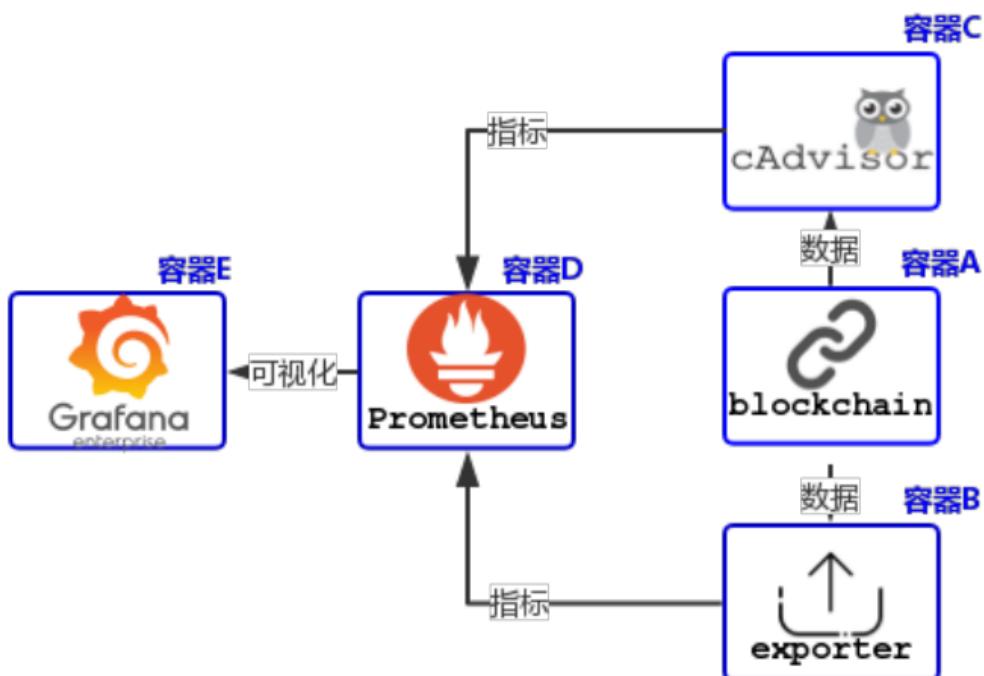


图 A1 系统结构图

### 3. 系统运行环境

系统运行环境包括容器的安装与使用、系统配置与软件配置。

### 3. 1 Docker 安裝

Docker 的安装方式与宿主机操作系统有关：

## 1) Windows

Docker for Windows 的当前版本运行在 64 位 Windows 10 Pro, 专业版、企业版和教育版上，在安装之前需要开启电脑的虚拟化，此项可在任务管理器中的性能板块查看状态，同时需要在控制面板中将 Hyper-V 功能启动。家庭版则不能直接使用 Docker for Windows，需要安装 Docker Toolbox，工具箱会将需要的所有环境自动安装，随后使用 Docker Quickstart Terminal 即可，界面如图 A2。

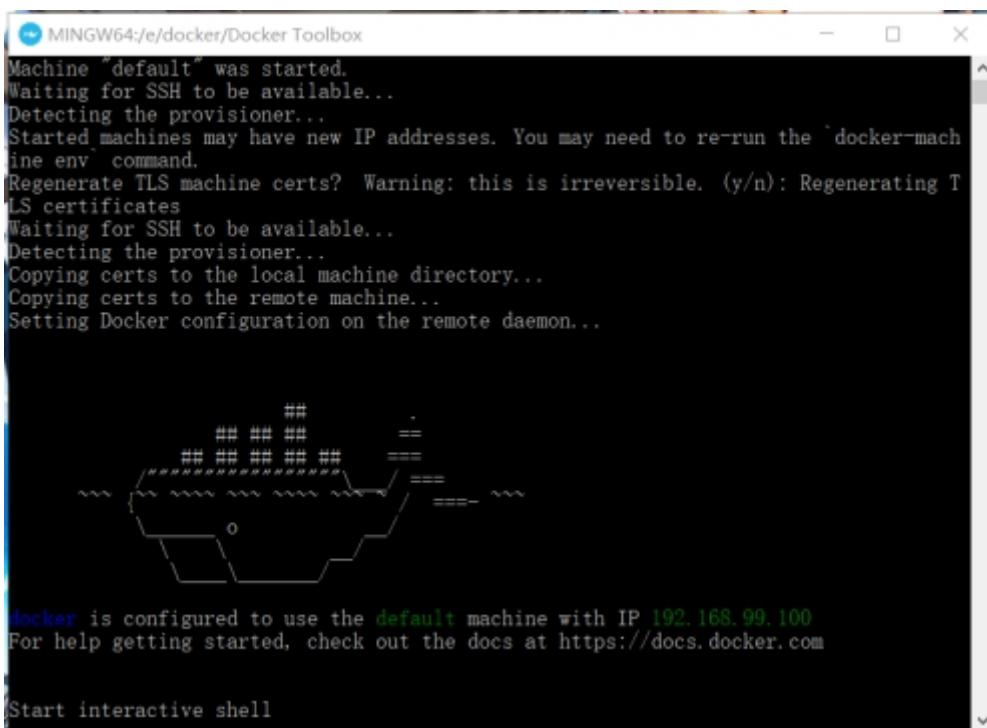


图 A2 Docker Toolbox

## 2) Linux

打开终端，使用命令 `sudo apt-get update` 更新 apt 后，键入 `sudo apt-get install -y docker.io`，安装完成后，可以使用 `docker version` 来查看是否安装成功，注意在使用 Docker 命令前需要 `su` 来获取 root 权限。随后使用 `systemctl status docker` 来查看 Docker 服务是否运行，若是 `inactive` 状态，则需要 `systemctl start docker` 启动 Docker 服务，启动后显示图 A3 的状态，`systemctl enable docker` 命令则可以设置开机自启。

```
root@ruanruanruan-VirtualBox:/home/ruanruanruan# systemctl status docker
● docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor pr
  Active: active (running) since 日 2020-03-29 17:28:58 CST; 27min ago
    Docs: https://docs.docker.com
```

图 A3 Docker Service 正在运行

需要注意的是，Docker 所有命令的使用都需要获取管理员权限。

### 3.2 系统环境配置

#### 1) python 环境：

本文使用 Ubuntu16.04-1 操作系统，系统默认指向 python2，为了同时使系统底层功能正常使用 python2 并且使用 python3 的依赖包，需要通过以下步骤，设置系统的 python 解释器指向 python3：

①sudo cp /usr/bin/python /usr/bin/python\_bak，将 python2 进行备份，若之后出现错误或需要使用 python2 时，可重新将 python 命令指向此备份。

②sudo rm /usr/bin/python，删除原先指向 python2 的文件。

③sudo ln -s /usr/bin/python3.5 /usr/bin/python，使 python 默认指向 python3。

使用 python -v 查看更改设置后的版本是否为 3.x。

#### 2) go 环境：

①使用 add-apt-repository ppa:longsleep/golang-backports 获取最新软件包，并添加至主机 apt 库。

②使用 apt-get update 对 apt 库进行更新。

③通过 sudo apt-get install golang-go 安装最新版 go。

④键入 go version 查看版本，鉴定安装是否成功。

### 3.3 应用安装及配置

#### 1) cAdvisor：

使用 docker pull google/cadvisor:latest 进行安装即可。

在 cAdvisor 中有几个常用的重要指标用来监控容器对资源的使用情况，通过它们可以计算得到我们所需要的指标，对于以太坊监控，需要在指标名称后加上镜像名称”ethereum-go/client-v3.0”来专门获取以太坊数据：

①CPU 使用总时间 container\_cpu\_usage\_seconds\_total

②内存占用 container\_memory\_usage\_bytes

③磁盘读字节数 container\_fs\_reads\_bytes\_total

④磁盘写字节数 container\_fs\_writes\_bytes\_total

⑤网络上传的总字节数 container\_network\_transmit\_bytes\_total

⑥网络下载的总字节数 container\_network\_receive\_bytes\_total

⑦容器最后存在的时间 container\_last\_seen

## 2) Prometheus:

使用 docker pull prom/prometheus:latest 进行安装，

Prometheus 根据配置文件来抓取目标性能指标，因此在使用前，需要在宿主机上创建配置文件并编辑。步骤如下：

①使用 touch /usr/local/prometheus.yml 新建配置文件。

②通过 vi /usr/local/prometheus.yml 启动编辑器，并按下键盘上 insert 键进行文件编辑。

③将表 A4 中内容复制进去，按下 esc 键，键入:wq，表示写文件，并退出，即可完成编辑。

④规则文件的编写流程相同，内容见表 A5。若 vim 编辑器不能正常使用（版本过低），可使用 apt-get remove vim-common 移除旧版后，使用 apt-get install vim 重新下载。

表 A4 rule.yml 文件

```
groups:
```

```
- name: Test_Ethereum
```

```
rules:
```

```
- expr: |
```

```
TPS_Ethereum*5/sum( increase(container_cpu_usage_seconds_total[5s])*2.5 )
record:TPC_Ethereum
```

```
- expr: |
```

```
TPS_Ethereum*5/(sum(increase(container_fs_reads_bytes_total[5s]))+sum(increase(container_fs_writes_bytes_total[5s])))*1024
record: TPPIO_Ethereum
```

```
- expr: |
```

```
TPS_Ethereum*5/sum( increase(container_network_receive_bytes_total[5s])+increase(container_network_transmit_bytes_total[5s]))*1024
record: TPN_Ethereum
```

```
- expr: |
```

```
TPS_Ethereum*5/sum( sum( increase(container_memory_usage_bytes[5s]) )/2+sum(container_memory_usage_bytes) )*1024*1024/5
record: TPMS_Ethereum
```

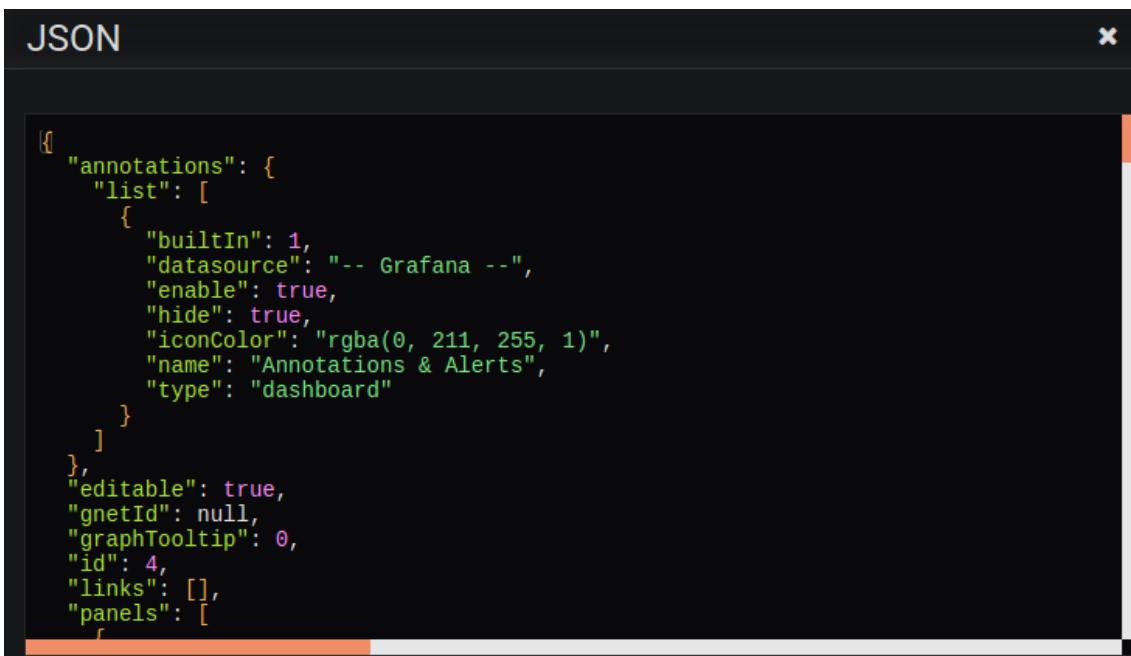
表 A5 prometheus.yml 文件

```
global:                                #全局定义
  scrape_interval: 1s                  #抓取数据的时间间隔
  evaluation_interval: 1s              #规则警报计算的时间间隔
rule_files:
  - "rule.yml"
scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']
  - job_name: Blockcharin_Ethereum
    static_configs:
      - targets: ['49.235.69.130:8080']
      - targets: ['49.235.69.130:12345']
      - targets: ['49.235.69.130:12580']
```

配置文件编写好后可以使用 `promtool check /usr/local/prometheus.yml` 来验证文件中是否有格式或语法错误。

### 3) Grafana

使用 `docker pull grafana/grafana:latest` 安装即可。在 Grafana 中使用的仪表板的部分 JSON 文件内容如下图 A6 所示，通过将 JSON 文件导入 Grafana 可以方便地实现图表的移植。



```
{
  "annotations": {
    "list": [
      {
        "builtin": 1,
        "datasource": "-- Grafana --",
        "enable": true,
        "hide": true,
        "iconColor": "rgba(0, 211, 255, 1)",
        "name": "Annotations & Alerts",
        "type": "dashboard"
      }
    ],
    "editable": true,
    "gnetId": null,
    "graphTooltip": 0,
    "id": 4,
    "links": [],
    "panels": [
      {
        "grid": {
          "grid": true,
          "gridPos": {
            "x": 0,
            "y": 0,
            "w": 12,
            "h": 12
          },
          "x": 0,
          "y": 0
        },
        "gridPos": {
          "x": 0,
          "y": 0,
          "w": 12,
          "h": 12
        },
        "height": 300,
        "title": "Annotations & Alerts"
      }
    ]
  }
}
```

图 A6 Dashboard-JSON 文件

## 4. Ethereum 测试流程

本节将从区块链网络搭建开始，到启动监控系统，到获取到数据，进行完整而具体的介绍。

### 4.1 网络搭建

在使用说明书中，仅介绍两个节点的区块链网络搭建方法，在两台主机上都需要进行创世区块的配置，其大致步骤如下：

- ① 使用 docker pull ethereum/client-go 安装以太坊客户端。
- ② 键入 docker run -it -p 30303:30303 -p 8545:8545 --entrypoint /bin/sh ethereum/client-go:v3.0 来启动以太坊客户端，并打开 30303 端口使节点互连，打开 8545 端口打开 RPC 调用接口。
- ③ 创建数据存储目录 mkdir -p /home/ubuntu/nodeA/data0 并进入 cd/home/Ubuntu /nodeA，随后新建 genesis.json 文件并进行创世区块的配置。
- ④ 使用 geth --datadir data0/ --networkid 66 --nodiscover console 启动以太坊服务
- ⑤ 使用 admin.addPeer("enode://d9fc148c9808fbfee7954bd3324bfdff42777de7d2545ac3c2357f05939dbd8ee153cd32320a05f4dbf18138007f743f3a2097b62e71c3c47bb3cf1559dd1328@ip:30303") 将节点连接起来，其中字符串为节点的 enode 标识，IP 为节点所在的主机地址，加入后可使用 admin.peers 来查看加入网络的节点信息。
- ⑥ 使用 personal.newAccount("account1") 和 personal.newAccount("account2") 新

建两个账户，并通过 `eth.coinbase` 来设置挖矿账户，若不设置账户，则无法进行挖矿。

⑦使用 `miner.start()` 开始挖矿即可产生区块。

更详细的过程可以参考有关联盟链搭建的博客：<https://blog.51cto.com/clovemfong/2280872>。

## 4.2 获取区块链数据

### 1) 交易生成器

首先要使用交易生成器，不断产生交易，于是节点将交易打包进区块中。交易生成器的 python 代码如下表 A7 所示。

表 A7 交易生成器源码

```
from flask import Flask, request, jsonify
import requests
import json
import time
from numpy import *
#发送交易
transaction = {
    "jsonrpc":"2.0",
    "method":"eth_sendTransaction",
    "params": [
        {
            "from": "0x85a3156c69c79bd363e676e04b954da8b4b6713e",
            "to": "0xbd5a7fb2ff97528afc547896606c11db2830a48c",
            "gas": "0xc000",
            "gasPrice": "0xa000",
            "value": "0x1",
            "data": ""
        }
    ],
    "id":1
}
headers = {'Content-Type': 'application/json'}
for i in range (1,100000000):
    times = time.time()
    r=requests.post('http://localhost:8545',data=json.dumps(transaction),headers=hea
```

```

ders)

delay = time.time() - times

response_str = r.text

response_json = json.loads(response_str)

print(r.text)

```

使用 python3 rpcdelay.py 命令启动，在命令行可见如下图 A8 的输出：

```

["jsonrpc":"2.0","id":1,"result":"0x167bbe78a5a4efff45f0da7eb81fed3a7133913bf1ecf66dd4cecf7e1c0d543"}
{"jsonrpc":"2.0","id":1,"result":"0xc6642519a6b1250d9efb8f5981049bdb23185aa6d8b9e1ee1e5844cf8486797"}
{"jsonrpc":"2.0","id":1,"result":"0x194c2dfb7e18cbe89c80cda93e089f3569d613bee88ed27ec8488d5955cf60d2"}
{"jsonrpc":"2.0","id":1,"result":"0x998516f2a3ec8593f4a54226086bbe5b0804f73575ee3454e2b36e8fc9eeab3c"}
{"jsonrpc":"2.0","id":1,"result":"0x4e2fafaf59ed250b2fd505feb7bb97995a5b80a728b3a2f08044b29302ac8ce89"}
{"jsonrpc":"2.0","id":1,"result":"0xb705e12f9bd06b34bf69234a4ea3f2c84a2b91ed0fcf1fa4a80de54cc15085a"}
{"jsonrpc":"2.0","id":1,"result":"0x743ee8e3f29a54647fdbd5dd78901a49b7dfe7b5beb9720f68ce24ea4b30b620"}

```

图 A8 持续发送交易

## 2) 计算吞吐量

吞吐量计算并输出为指标的 python 脚本代码如下表 A9：

表 A9 吞吐量计算关键源码

```

import requests

import json

import time

import numpy as np

from prometheus_client import start_http_server,Gauge


res=Gauge('TPS_Ethereum','throughput of ethereum')

start_http_server(12580)

TPS_list=[]

for i in range(100000):

    eth_blockNumber={"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":1}

    getBlockTransactionCountByNumber={"jsonrpc":"2.0","method":"eth_getBlock

    TransactionCountByNumber","params":[""],"id":1}

    headers = {'Content-Type': 'application/json'}

    r=requests.post('http://localhost:8545',data=json.dumps(eth_blockNumber),head

```

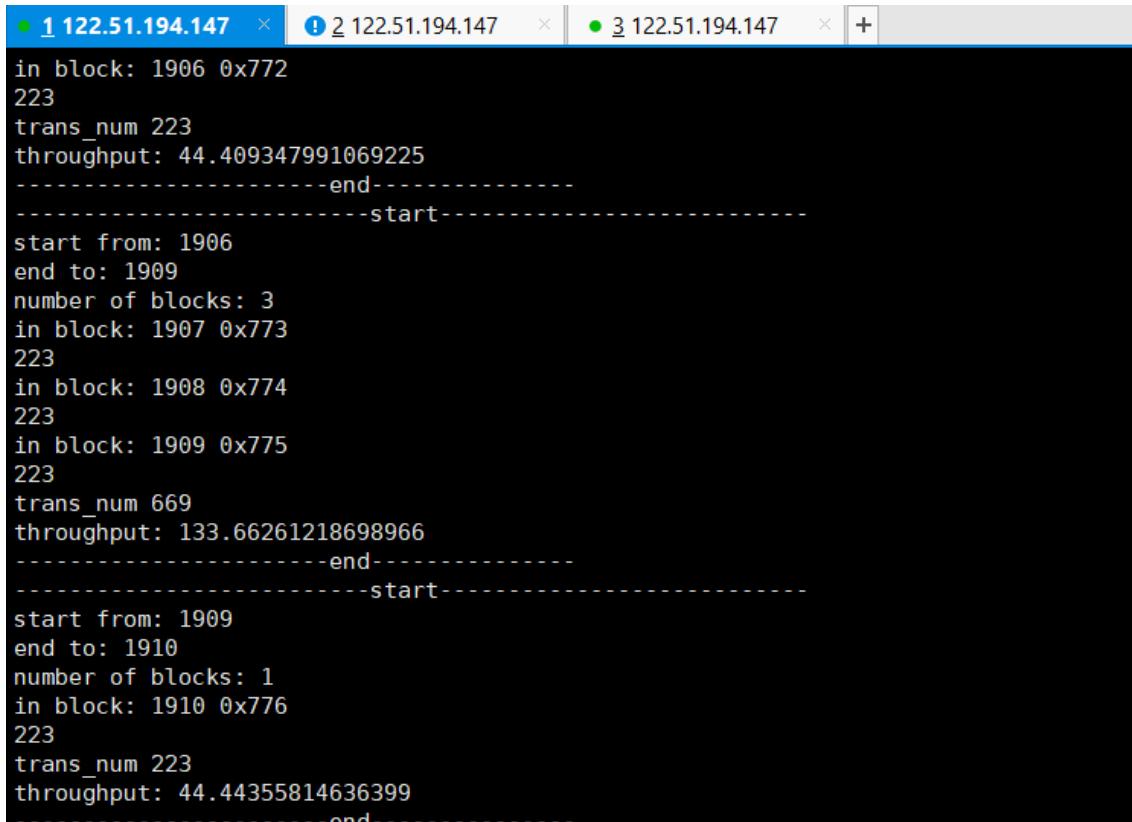
```

ers =headers)
response_str = r.text
response_json = json.loads(response_str)
result = int(response_json['result'],base=16)
times = time.time()
time.sleep(5)
w=requests.post('http://localhost:8545',data=json.dumps(eth_blockNumber),he
ders=headers)
response_str = w.text
response_json = json.loads(response_str)
result1 = int(response_json['result'], base=16)
sum = result1 - result
t1 = time.time() - times
print("-----start-----")
print("start from:", result)
print("end to:", result1)
print("number of blocks:", sum)
list = []
for i in range(0, sum):
    result += 1
    print("in block:", result, hex(result))
    result = hex(result)
    result = str(result)
    getBlockTransactionCountByNumber["params"]=[result]
    w=requests.post('http://localhost:8545',data=json.dumps(getBlockTran
sactionCountByNumber), headers=headers)
    count_str = w.text
    count_str = json.loads(count_str)
    count_number = int(count_str['result'], base=16)
    print(count_number)
    list.append(count_number)
    result = int(result, base=16)
print("trans_num",np.sum(list))
print("throughput:",np.sum(list)/t1)
res.set(np.sum(list)/t1)
TPS_list.append(np.sum(list)/t1)

```

```
print("-----end-----")
print(TPS_list)
```

使用 python3 throughput.py 启动脚本后，命令行会显示区块信息，如图 A10：



```
● 1 122.51.194.147 ✘ | ⓘ 2 122.51.194.147 ✘ || ● 3 122.51.194.147 ✘ [+]
in block: 1906 0x772
223
trans_num 223
throughput: 44.409347991069225
-----end-----
-----start-----
start from: 1906
end to: 1909
number of blocks: 3
in block: 1907 0x773
223
in block: 1908 0x774
223
in block: 1909 0x775
223
trans_num 669
throughput: 133.66261218698966
-----end-----
-----start-----
start from: 1909
end to: 1910
number of blocks: 1
in block: 1910 0x776
223
trans_num 223
throughput: 44.44355814636399
-----end-----
```

图 A10 吞吐量计算

### 3) 计算时延

计算交易时延并输出为指标的 python 脚本代码如下表 A11：

表 A11 时延计算关键源码

```
from flask import Flask, request, jsonify
import requests
import json
import time
from numpy import *
from prometheus_client import start_http_server,Gauge,Histogram
#resres=Gauge('Delay_Ethereum','时延',['transactions','blockchain'])
resres=Gauge('Delay_Ethereum_gau','时延')
his=Histogram('Delay_Ethereum_his','时延直方图',buckets=(0,10,20,30,40,50,60,70,
80,90,100,110,120))
start_http_server(12345)
```

```
#the content of transaction sending API
transaction = {
    "jsonrpc": "2.0",
    "method": "eth_sendTransaction",
    "params": [
        {
            "from": "0x85a3156c69c79bd363e676e04b954da8b4b6713e",
            "to": "0xbd5a7fb2ff97528afc547896606c11db2830a48c",
            "gas": "0xc000",
            "gasPrice": "0xa000",
            "value": "0x1",
            "data": ""
        }
    ],
    "id": 1
}
TransByhash = {"jsonrpc": "2.0", "method": "eth_getTransactionByHash", "params": [], "id": 1}
headers = {'Content-Type': 'application/json'}
list = []
for i in range(1, 100000):
    times = time.time()
    r = requests.post('http://localhost:8545', data=json.dumps(transaction), headers=headers)
    print(r.text)
    response_str = r.text
    response_json = json.loads(response_str)
    TransByhash["params"] = [response_json['result']]
    print("hash=", response_json['result'])

#通过 hash 重复查询区块上是否存在此交易直到交易被成功打包
res = None
while (res is None):
    w = requests.post('http://localhost:8545', data=json.dumps(TransByhash),
                      headers=headers)
    response_str1 = w.text
    response_json1 = json.loads(response_str1)
    res = response_json1['result']['blockNumber']
    time.sleep(0.1)
```

```

s= time.time() - times
list.append(s)
print(time.time() - times)
k=float(s)
resres.labels(transactions=response_json['result'],blockchain='ethereum').set(k)
#设置指标数值为延迟时间
resres.set(k)
his.observe(k)

```

使用 python3 confirm\_lantency.py 命令运行脚本后显示如下图 A12 所示：

```

root@VM-16-17-ubuntu:/home/ubuntu# python3 confirm_lantency.py
{"jsonrpc":"2.0","id":1,"result":"0xa5561a9ae283407d59b1705e8dfb6c16d4ee2c8e09ec2a08bcb3aad9f17f3e48"}
hash= 0xa5561a9ae283407d59b1705e8dfb6c16d4ee2c8e09ec2a08bcb3aad9f17f3e48
95.19801425933838
{"jsonrpc":"2.0","id":1,"result":"0x313018bb8ff148dc22d4c5e86ae623501d8d92b33f45e5a41c11b96d79d46700"}
hash= 0x313018bb8ff148dc22d4c5e86ae623501d8d92b33f45e5a41c11b96d79d46700
139.5269181728363

```

图 A12 时延计算

#### 4) 容器资源监控

通过上述脚本获取完区块链吞吐量和时延后，需要通过 `cadvisor` 来监测容器对资源的使用情况，`cadvisor` 与测试脚本应一同运行，数据库需要同时收集这两者的数据来完成区块链的测试。`cadvisor` 安装完成后使用下列命令启动：

```

docker run \
--volume=/:/rootfs:ro \
--volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \
--volume=/var/lib/docker:/var/lib/docker:ro \
--volume=/dev/disk:/dev/disk:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor

```

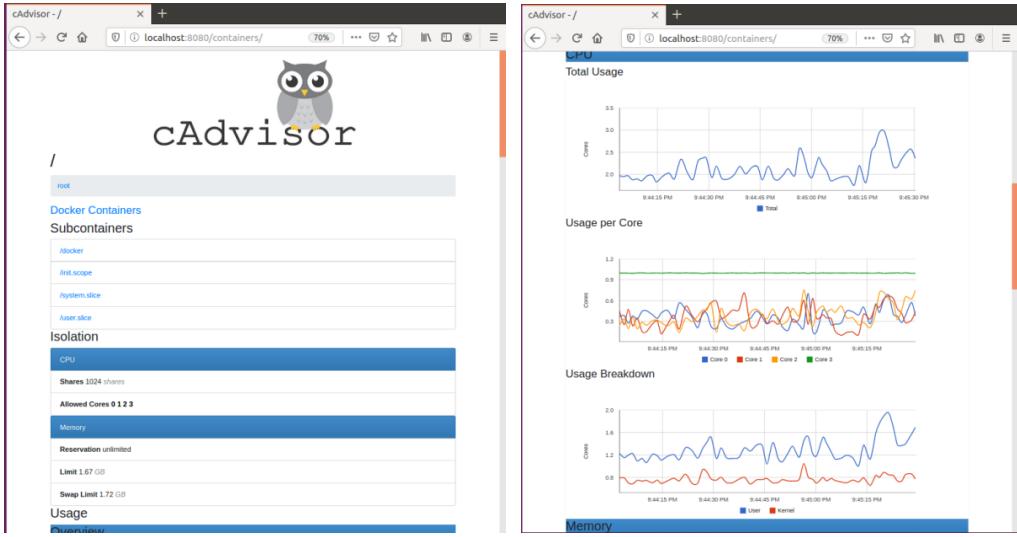


图 A13 cAdvisor 主界面

随后可以打开浏览器，进入 localhost:8080 页面，查看 cAdvisor 是否正常运行，如图 A13，或进入 localhost:8080/metrics 查看是否成功显示 metrics，若有如下图 A14 的数据显示，且数值没有异常，表明 exporter 正常工作。

```
# HELP cAdvisor_version_info A metric with a constant '1' value labeled by kernel version, OS version, docker version, cAdvisor version & cAdvisor revision.
# TYPE cAdvisor_version_info gauge
cAdvisor_version_info{cAdvisorRevision="9949c822",cadvisorVersion="0.32.0",dockerVersion="18.09.7",kernelVersion="4.15.0-91-generic",osVersion="Alpine Linux v3.7"} 1
# HELP container_cpu_load_average_10s Value of container cpu load average over the last 10 seconds.
# TYPE container_cpu_load_average_10s gauge
container_cpu_load_average_10s{id="/",image="",name=""} 0
container_cpu_load_average_10s{id="/docker",image="",name=""} 0
container_cpu_load_average_10s{id="/dockersocket",image="c898d4699afde8703764d78f526c6ebfb7b485c14a97e58519a79200027",name="cAdvisor"} 0
container_cpu_load_average_10s{id="/init.slice",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/wait-online.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/NetworkManager",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/accounts_daemon.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/acpid.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/alsa-restore.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/avahi-daemon.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/cron.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/ptp-daily-upgrade.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/avahi-daemon.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/cgroups-mount.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/colorl.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/console-setup.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/containerd.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/cronv1.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/cups-braved.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/cups.service",image="",name=""} 0
container_cpu_load_average_10s{id="/system.slice/dbus.service",image="",name=""} 0
```

图 A14 cAdvisor 的指标显示界面

关闭 cAdvisor 命令为 docker stop id(id 指容器标识，可用 docker ps 命令查询)，关闭后容器不会消失，再次启动时使用 docker start id 即可，若仍使用本节开头的命令，则会提示容器号已被占用，无法运行 cAdvisor。

需要注意的是，由于 cAdvisor 对磁盘读写的监控可能存在问题，如果磁盘读写总量长期保持不变，可以下载 node\_exporter，对其进行 9100:9100 的端口映射，并添加至数据库配置文件中，使用它所提供的主机磁盘读写数据。

### 4.3 启动数据库

在这一步启动时序数据库 Prometheus。按前述 3.3 要求写好配置文件后，便可以启动数据库，若需要进行数据持久化，则使用以下命令启动数据库，注意此处数据保存路径 tmp 文件夹为临时文件夹，系统重启后数据就会擦除，可根据使用需

要来更改。

```
docker run -p 9090:9090 \
    #tmp 为 linux 临时文件夹，冒号前为自定义存储路径
    -v /tmp/prometheus-data:/etc/prometheus-data \
    /usr/local/prometheus.yml:/etc/prometheus/prometheus.yml \
    #数据存储时间，此处为一天
    -storage.local.retention 168h0m0s \
    #保存的 chunk 最大大小，默认为 1G，即 1048576
    -storage.local.memory-chunks=50502740 \
    #数据库进行采集和查询的时候，需要大量全局锁，如果分配的不够会导致
    #延迟，因此在高并发时可以适当提高此项
    -storage.local.num-fingerprint-mutexes=300960
    prom/prometheus
```

测试时若不需要存储数据，可以仅用以下命令： docker run –name prometheus -p 9090:9090 -v /usr/local/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus。启动后通过本机 localhost:9090 进入数据库查询页面，如图 A15 所示。

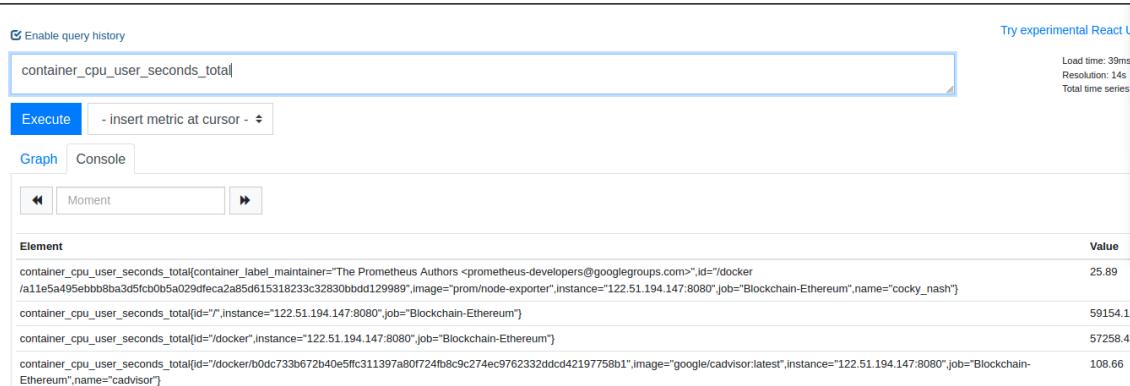


图 A15 Prometheus 首页

每次登录云服务器时，其 IP 会改变，因此脚本输出数据的 IP 以及 cAdvisor 的 IP 随之发生变化，此时需要通过 Prometheus 中的 yml 配置文件更改 exporter 的端口或 IP 地址，并需要重新加载文件以应用新的数据端口。可采用热更新方法，在数据库运行的同时更新配置：

- ①在 Linux 命令行中使用 ps -ef|grep prom/prometheus 查询数据库的进程号。
- ②使用 kill -HUP pid 即可完成热更新。
- ③在数据库运行日志下可以看到配置文件重新加载的提示，如图 A16。

```

level=info ts=2020-03-25T09:53:56.558Z caller=main.go:762 msg="Completed loading of configuration file" filename=/etc/prometheus/prometheus.yml
level=info ts=2020-03-25T09:53:56.558Z caller=main.go:617 msg="Server is ready to receive web requests."
level=info ts=2020-03-25T09:58:58.907Z caller=main.go:734 msg="Loading configuration file" filename=/etc/prometheus/prometheus.yml
level=info ts=2020-03-25T09:58:58.914Z caller=main.go:762 msg="Completed loading of configuration file" filename=/etc/prometheus/prometheus.yml

```

图 A16 热更新加载配置

数据库配置正常加载后，需要验证是否连接上所有 exporter，可进入 localhost:9090 /targets 页面查看，如图 A17，States 项为 UP 表示连接成功，点击 Endpoint 可查看端口内的数据。

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://122.51.194.147:12345/metrics	UP	instance="122.51.194.147:12345" job="Blockchain-Ethereum"	688ms ago	50.49ms	
http://122.51.194.147:12580/metrics	UP	instance="122.51.194.147:12580" job="Blockchain-Ethereum"	878ms ago	48.35ms	
http://122.51.194.147:8080/metrics	UP	instance="122.51.194.147:8080" job="Blockchain-Ethereum"	247ms ago	119.1ms	
http://122.51.194.147:9100/metrics	UP	instance="122.51.194.147:9100" job="Blockchain-Ethereum"	372ms ago	36.91ms	

图 A17 Targets 状态

#### 4.4 数据可视化

使用 docker run --name grafana -p 12200:3000 grafana/grafana 启动 Grafana，使用账号 admin，密码 admin 进入，随后配置数据源，选择 Prometheus 并键入其 web 页面地址，如图 A18 所示。

图 A18 数据资源配置

若直接使用写好的仪表板，则通过 Import 选项导入仪表板 JSON 文件即可。若需要个性化设置，可以通过增减 Panel 来更改显示面板，下面将介绍在以太坊测试中所使用的两种图像。

1) Graph: 图表为常见的曲线图，是 Grafana 最主要的图形面板可以直观地展现系统性能在一段时间内的变化情况，并对未来走向进行预测。如图 A19，在 General 选项下可以设置图标的标题和描述，Metrics 用于便捷设置数据源和数据类型，Axes 用于调整横纵坐标相关的显示，而 Display styles 用于控制可视化的特性，如 Bar 将数据显示为条形图，Lines 为曲线图，Points 为显示样本点，通过显示样式可以美化图形。

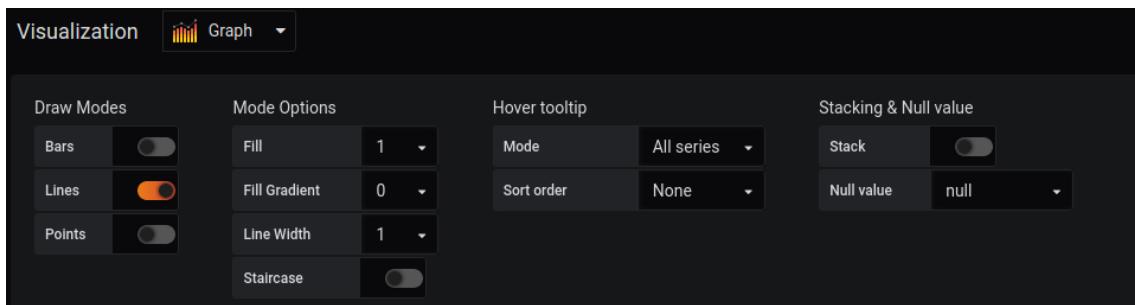


图 A19 Graph 设置

在 Graph 中使用的查询语句如下：

- ①吞吐量：TPS\_Ethereum
- ②时延：Delay\_Ethereum\_his\_bucket
- ③TPC：  

$$\text{sum}(\text{TPS\_Ethereum}) * 5 / \text{sum}(\text{increase}(\text{container\_cpu\_usage\_seconds\_total}\{\text{image}=\text{"ethereum/client-go:v3.0"}\}[5s])) * 2.5$$
- ④TPN：  

$$\text{sum}(\text{TPS\_Ethereum}) * 5 / \text{sum}(\text{increase}(\text{container\_network\_receive\_bytes\_total}\{\text{image}=\text{"ethereum/client-go:v3.0"}\}[5s]) + \text{increase}(\text{container\_network\_transmit\_bytes\_total}\{\text{image}=\text{"ethereum/client-go:v3.0"}\}[5s])) * 1024$$
- ⑤TPMS：  

$$\text{sum}(\text{TPS\_Ethereum}) * 5 / \text{sum}(\text{sum}(\text{increase}(\text{container\_memory\_usage\_bytes}\{\text{image}=\text{"ethereum/client-go:v3.0"}\}[5s])) / 2 + \text{sum}(\text{container\_memory\_usage\_bytes}\{\text{image}=\text{"ethereum/client-go:v3.0"}\})) * 1024 * 1024 / 5$$
- ⑥TPDIO：  

$$\text{sum}(\text{TPS\_Ethereum}) * 5 / ((\text{sum}(\text{increase}(\text{container\_fs\_reads\_bytes\_total}\{\text{image}=\text{"ethereum/client-go:v3.0"}\}[5s])) + \text{sum}(\text{increase}(\text{container\_fs\_writes\_bytes\_total}\{\text{image}=\text{"ethereum/client-go:v3.0"}\}[5s]))) / 1024)$$

2) Gauge: 计量器多用于显示 gauge 类型，可动态增减的数据。如图 A20，在显示选项 Display options 中，Show 表示数据显示类型，包括计算类型和直接展示两种，Calculate 用于设置计算选项，如平均值、最大值、最小值，Labels 用于设置是否在仪表盘上显示阈值，Threshold 用于控制阈值大小。

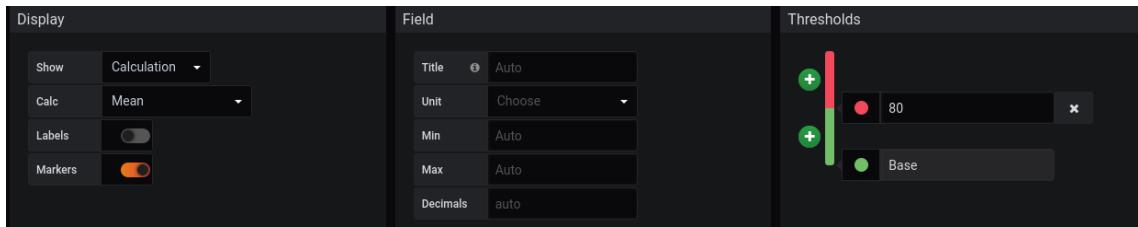


图 A20 Gauge 设置

在测试中，本文选用 Calculate 中的 mean 类型，用于展示数据的均值，所用的查询语句除了时延测试不同之外，其他的与 Graph 中语句相同。此处时延使用 Histogram 来统计，将不同区间上的时延分别统计，语句为 Delay\_Ethereum\_his\_bucket。

3) 配置完成后，回到仪表板，便可以进行网络监控了，同时需要在右上角刷新选项中选择 5s 或 10s 来定时刷新页面，使图像动态变化，在刷新选项左边也可以选择监控的时间间隔。在某次测试中，对以太坊的性能监控结果如下图 A21 和图 A22：

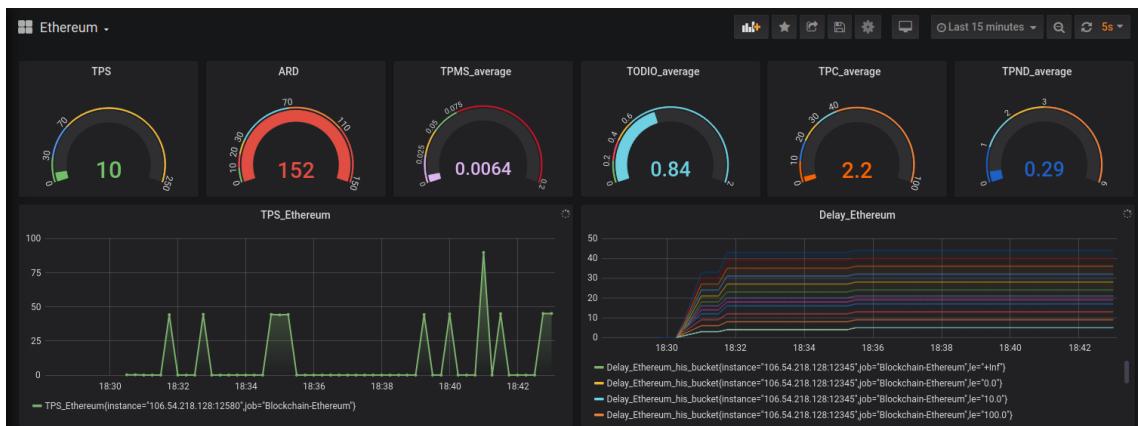


图 A21 以太坊监控-1

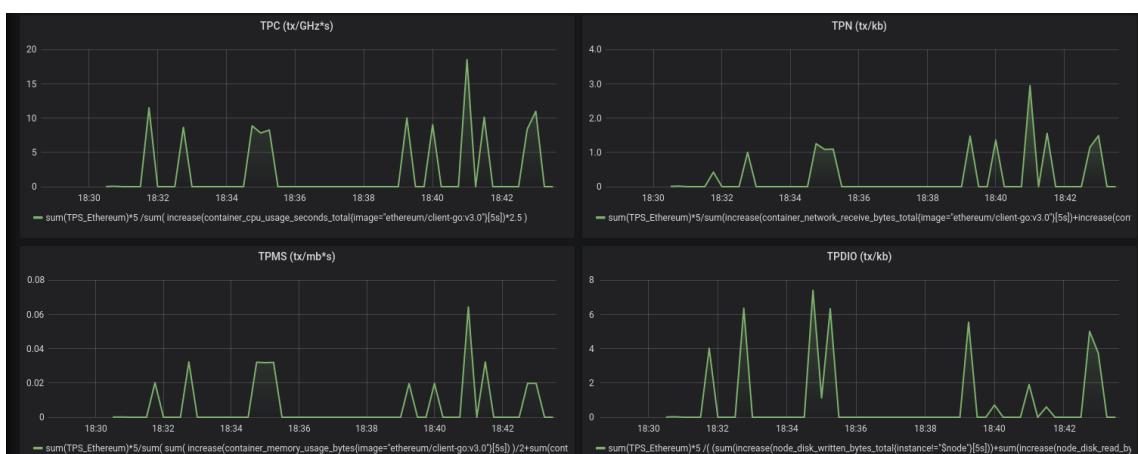


图 A22 以太坊监控-2