1 What is Starknet

区块链旨在实现三个核心属性：安全性、去中心化和可扩展性。在区块链世界中，一个众所周知的三难困境是，在给定系统中只能同时实现其中两个，不可避免地需要在第三个上做出妥协。以太坊更加重视安全性和去中心化，这影响了其可扩展性。以太坊用户数量的增长导致交易速度缓慢和汽油价格高昂，阻碍了以太坊的广泛采用。

如何才能在不损害以太坊安全性和去中心化的情况下使以太坊具有可扩展性？这就是 Starknet 有效性汇总。将以太坊和 Starknet 结合起来可以实现巨大的可扩展性。

Starknet 通过将交易处理移出以太坊主网（我们称之为链下）来实现规模化，同时保留链上交易的摘要。交易被分批分组到区块中，在链下处理，然后汇总成单个链上交易。由于交易发生在链外，因此确保交易及其执行的完整性而无需重新执行它们至关重要。 Starknet 通过采用 STARK（可扩展、透明的知识论证）证明进行可验证计算来解决这个问题。然后，Starknet 仅将有关该块的基本信息和证明传输到以太坊，并以最少的计算量对其进行验证。

1.1The Starknet Prover

证明者通过生成 STARK 证明来建立区块交易的数学有效性，从而确保其完整性。块被组织成组并同时处理。在此阶段中，证明者记录事务执行的每个步骤，创建所谓的执行跟踪，并跟踪系统状态的结果变化（称为状态差异）。

然后，算法会仔细地分解并混合来自执行跟踪的数据。此步骤将任何问题带到最前沿，因为即使是单个不良数据实例也会污染整个扩展数据集，从而使任何问题都不容忽视。然后证明者从这些放大的数据中选择一组随机样本来创建 STARK 证明。这个 STARK 证明断言了数千笔交易的有效性。

1.2以太坊上的安全结算

STARK 证明和 State Diff 作为交易传输到以太坊，以太坊节点接受交易并解包证明和 State Diff。这些解压后的组件由两个以太坊智能合约处理：Verifier 和 Starknet Core。验证者合约分解证明并分析其中的样本。证明样本中任何有问题数据的暗示都会导致验证者立即拒绝。一旦证明的有效性得到确认，它就会进入 Starknet Core 智能合约。

核心合约验证证明的真实性并确认收到状态差异，随后更新以太坊区块链上的 Starknet 状态。然后将更新后的状态添加到以太坊区块中，分布在节点网络上以进行验证和投票。当该区块积累足够的选票时，它就会达到“最终确定”状态，巩固其作为以太坊不可篡改部分的地位。

2 The Starknet Network

2.1 前言

从历史上看，货币、财产权和社会地位头衔等社会角色一直受协议和登记管理。他们的价值源于对其诚信的广泛接受的理解。这些职能主要由中央实体监督，容易面临腐败、机构冲突和排斥等挑战（Eli Ben-Sasson、Bareli、Brandt、Volokh，2023）。

中本聪的创造比特币为这些功能引入了一种新颖的方法，称为完整性网络。这是一个社会角色的基础设施：

1.由公共协议公开描述。
2.在广泛、包容的点对点网络上运营。
3.公平、广泛地分配价值，以维持社会对其诚信的共识.

比特币解决了货币功能，而以太坊则将其扩展为包括可以通过计算机编程定义的任何功能。两者都面临着平衡可扩展性和去中心化的挑战。这些完整性网络通常更倾向于包容性而不是容量，确保即使资源有限的人也可以验证系统的完整性。然而，这意味着他们难以满足全球需求。

在不断发展的技术领域，定义像“区块链”这样多方面的术语可能具有挑战性。根据目前的理解和应用，区块链可以具有以下三个属性（Eli Ben-Sasson，2023）：

公共协议：

区块链的基础依赖于公开可用的协议。这种透明度确保任何感兴趣的各方都可以了解其运作方式，从而培养信任并实现更广泛的采用。

开放的 P2P 网络：

区块链不依赖集中式实体，而是在点对点 (P2P) 网络上运行。这种去中心化的方法确保操作分布在各个参与者或节点上，使系统对故障和审查更具弹性。

价值分配：

区块链运营的核心是它奖励运营商的方式。该系统以广泛且公平的方式自主分配价值。这种激励不仅激励参与者维护系统的完整性，而且确保了更广泛的社会共识。

虽然这些属性抓住了许多区块链的本质，但随着技术的成熟和新应用的发现，该术语的定义可能需要完善。在这个充满活力的环境中，进行持续对话和重新审视定义至关重要。

2.2 Starknet Definition

Starknet 是一个第 2 层网络，它使用 zk-STARKs 技术使以太坊交易更快、更便宜、更安全。将其视为以太坊之上的增强层，针对速度和成本进行了优化。

Starknet 弥合了可扩展性和广泛共识之间的差距。它集成了一个数学框架来平衡容量和包容性之间的平衡。其完整性取决于简洁、透明的计算完整性证明的稳健性。这种方法可以让强大的运营商增强 Starknet 的能力，确保每个人都可以使用通用工具验证 Starknet 的完整性（Eli Ben-Sasson、Bareli、Brandt、Volokh，2023）。

2.2.1 Key Features

成本低：

Starknet 上的交易成本低于以太坊。 Volition 和 EIP 4844 等未来的更新将使其更加便宜。

开发人员友好：

Starknet 让开发人员可以使用其母语 Cairo 轻松构建去中心化应用程序。

速度和效率：

即将发布的版本旨在使交易更快、更便宜。

CVM：

通过 Cairo，Starknet 在自己的虚拟机上运行，称为 Cairo VM (CVM)，这使我们能够超越以太坊虚拟机 (EVM) 进行创新，并为去中心化应用程序创建新范例。

账户抽象：

在协议层面实现，有利于多样化的签名方案，同时确保用户安全和资产的自我托管。

Volition：

将于 2023 年第四季度在测试网上实施，允许开发人员调节以太坊 (L1) 或 Starknet (L2) 上的数据可用性。减少 L1 链上数据可以从根本上降低成本。

Paymaster：

Starknet 将允许用户选择如何支付交易费用，遵循 EIP 4337 中规定的准则，并允许交易指定一个特定的合约（Paymaster）来支付其交易。支持无gas交易，增强用户可访问性。

2.2.2 Governance

Starknet 基金会负责监督 Starknet 的治理。其职责包括：

```
1.管理 Starknet 的开发和运营
2.监督 Starknet DAO，促进社区参与
3.设置规则以维护网络完整性
```

成员可以通过对变更进行投票来影响 Starknet。流程如下：新版本在 Goerli 测试网上进行测试。成员们有六天的时间进行审查。提出快照提案，社区进行投票。多数赞成票意味着升级到主网。

SNIP: Starknet Improvement Proposals

SNIP 是 Starknet 改进提案的缩写。它本质上是一个蓝图，详细介绍了 Starknet 生态系统的拟议增强或更改。精心设计的 SNIP 包括变更的技术规范及其背后的原因。如果您提议 SNIP，您的工作就是争取社区支持并记录任何反对意见（更多详细信息请参见此处）。一旦 SNIP 获得批准，它就会成为 Starknet 协议的一部分。所有 SNIP 都可以在此存储库中找到。
SNIP 发挥着三个关键作用：

```
1.它们是提出新功能或更改的主要途径。
2.它们充当社区内技术讨论的平台。
3.他们记录了决策过程，提供了 Starknet 演变的历史视角。
```

对于在 Starknet 上进行构建的人来说，SNIP 不仅仅是建议，还是路线图。对于实施者来说，保留他们已执行的 SNIP 列表是有益的。这种透明度可以帮助用户评估特定实施或软件库的状态。

2.3 The Starknet Stack

Starknet 的构建模块吸引了广泛的应用程序和用例。它们是：STARK 证明、Cairo 编程语言和本机帐户抽象。随着 Starknet 在主网上升级到 v0.12.0，Starknet 成为 TPS 性能最高的 L2。我们预计 Starknet 相对于其他 L2（尤其是相对于 EVM 兼容的 L2）的性能优势会随着时间的推移而增长，因为 Starknet 不受 EVM 设计和实现所施加的遗留约束的束缚。

Decentralization 去中心化

Starknet Stack 正在迅速成为最去中心化的 L2 堆栈。无需许可的区块链专注于去中心化，作为实现网络安全和弹性的一种手段。 Starknet 基金会致力于为 Starknet 实现这一特性。

“A decentralized stack makes the network more secure, resilient, ransparent, scalable and innovative. No single point of failure, no dependency on a single entity, no black boxes and many more builders!”
————Nicolas Bacca,Co-Founder & CTO, Ledger

2.4 The What’s What of the Cairo World

2.4.1 介绍

为了解锁以太坊的安全和去中心化扩展，有效性汇总使批量交易的验证比简单的重新执行更加高效。第 2 层 (L2) 上的专用节点（称为排序器）将交易捆绑到新的 L2 区块中，而以太坊主网节点则以最小的努力确认这些交易。

Starknet 是一个利用 Cairo VM 的有效性汇总，专门设计用于优化有效性证明的效率。 Starknet 采用STARKs（可扩展、透明的知识论证）作为证明系统，能够为复杂的计算生成简洁的证明，从而大大降低链上验证过程的复杂性。

2.4.2 Cairo VM

为通用计算程序创建有效性证明需要深入掌握 STARK 背后的复杂数学原理。对于每个计算，构建代数中间表示 (AIR) 至关重要，它包含一组准确表示给定计算的多项式约束。 Cairo 最初被称为“CPU AIR”，是一个虚拟 CPU 和一个单一的 AIR，能够使用相同的“通用”AIR 描述任何计算。 Cairo VM 是专门为有效性证明系统量身定制的，不受 EVM（以太坊虚拟机）施加的限制

```
                        CairoVM                                        EVM
Purpose                  Proof Optimised                          VM        Blockch
Computational    model Von Neumann architecture                Stack-based machin
Basic type           Field elements                                 256-bit words
Memory model           Nondeterministic read-only memory        Read \ write memo
              (key in optimizing the proof for memory accesses)
```

2.4.3 CASM

CASM（Cairo Assembly）是Cairo VM运行的机器代码。 CASM 被转换为强制程序正确执行的多项式约束。 CASM 是生态系统中的关键组件，因为无论用户向 Starknet 测序器发送什么内容，所证明的都是正确的 CASM 执行。

2.4.4 Cairo Zero

Cairo Zero 于 2020 年发布，推出了世界上第一个图灵完备的语言，用于创建 STARK 可证明的程序，彻底改变了可验证的计算。开罗零号程序在本地编译到 CASM 中，然后发送到 Starknet 定序器。尽管具有开创性，但开罗零号由于其低级性质而具有陡峭的学习曲线，并且没有完全抽象出证明程序执行所需的底层加密原语。

2.4.5 Cairo

Cairo（现在 v2.1.1）克服了 Cairo Zero 的局限性，承诺更安全、更高效的合约编写。 Cairo 通过类似 Rust 的语法并抽象出 Cairo Zero 中存在的限制（例如一次写入内存），极大地改善了开发人员的体验。

Cairo 从 Rust 世界带来了现代编程概念，例如特征/实现、泛型、枚举匹配，而不会影响底层 CairoVM 带来的证明生成效率。

2.4.6 Sierra

与cairo一起到来的是Sierra。 Sierra 充当cario和 CASM 之间的中间代表。这个附加层确保用户代码在所有情况下都保持可证明。 Sierra 编译为“安全 CASM”，这是 CASM 的一个子集，保证对所有输入都可证明。用户代码和经过验证的代码之间的中间层对于保护 Starknet 定序器免受无法证明的事务形式的 DOS 影响至关重要。

Sierra 的一个也许令人惊讶的好处是，由于这种简单的中间表示，Starknet 定序器最终可以直接在本机硬件上运行，而不是通过 CairoVM。为了说明执行 Sierra 的排序器的强大功能，请考虑以下示例：可以使用 Sierra 中的类型信息来处理本机类型（例如 u32），而不是在 CairoVM 的素数字段中工作。

2.4.7 Conclusion

Cairo 建立在 CairoVM 奠定的基础上，彻底改变了可验证计算。凭借类似 Rust 的语法和现代编程语言功能，Cairo 极大地增强了开发人员体验，简化了合约编写并减少了出现错误的机会。开罗成为推动去中心化创新的强大工具。

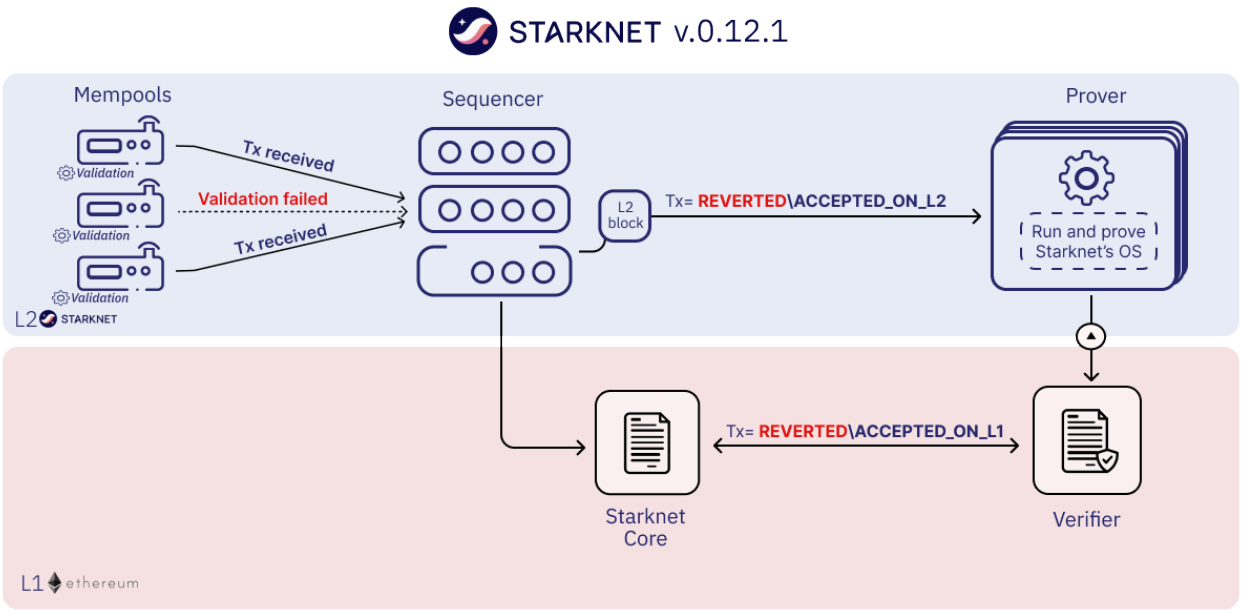
3 Architecture (架构)

对Starknet L2架构的介绍
Starknet 是一个协调系统，每个组件（排序器、证明器和节点）都扮演着特定但相互关联的角

色。尽管 Starknet 尚未完全去中心化，但它正在积极朝着这一目标迈进。对系统内角色和交互的理解将帮助您更好地掌握 Starknet 生态系统的复杂性。

3.1 交易

当网关（充当内存池）收到交易时，Starknet 的操作就开始了。该阶段也可以由定序器管理。该交易最初标记为“已接收”。然后，定序器将事务合并到网络状态中，并将其标记为“ACCEPTED_ON_L2”。最后一步涉及证明者，它在新区块上执行操作系统，计算其证明，并将其提交给第 1 层 (L1) 进行验证。



本质上，Starknet 的架构涉及多个组件:

1.排序器(sequencer)负责接收交易、对交易进行排序并生成区块。它的运作方式与以太坊或比特币中的验证器类似

2.证明者(prover)的任务是为创建的块和交易生成证明。它使用 Cairo 的虚拟机来运行可证明的程序，从而创建生成 STARK 证明所需的执行跟踪。

3.第 1 层 (L1)，在本例中为以太坊，托管能够验证这些 STARK 证明的智能合约。如果证明有效，Starknet 在 L1 上的状态根就会更新。

Starknet 的状态是通过 Merkle 树维护的全面快照，就像以太坊一样。这建立了有效性汇总的架构和每个组件的角色。

3.2 Sequencers (测序仪)

排序器是 Starknet 网络的支柱，类似于以太坊的验证器。他们将交易引入系统。
有效性汇总擅长将一些网络杂务（例如捆绑和处理交易）交给专门的参与者。这种设置有点像以太坊和比特币如何将安全性委托给矿工。测序和采矿一样，需要大量资源。

对于像 Starknet 这样的网络和其他利用有效性汇总的平台，也存在类似的情况。这些网络将交易处理外包给专门实体，然后验证他们的工作。有效性汇总上下文中的这些专门实体称为“排序器”。

排序器不像矿工那样提供安全性，而是提供交易能力。他们将多个交易排序（排序）到一个批次中，执行它们，并生成一个块，该块稍后将由证明者证明，并作为单个紧凑的证明（称为“汇总”）提交到第 1 层网络。换句话说，就像以太坊中的验证器和比特币中的矿工是保护网络安全的专门参与者一样，基于有效性汇总的网络中的排序器也是提供交易能力的专门参与者。

这种机制允许 Validity（或 ZK）汇总处理更大的交易量，同时维护底层以太坊网络的安全性。它在不影响安全性的情况下增强了可扩展性。

排序器遵循系统的事务处理方法：

```
1.排序：他们收集用户的交易并对它们进行排序（排序）。

2.执行：排序器然后处理这些事务。

3.批处理：交易以批次或块的形式分组在一起以提高效率。

4.区块生产：排序器生成包含批量已处理交易的区块。
```

排序器必须可靠且高度可用，因为它们的作用对于网络的平稳运行至关重要。他们需要功能强大且连接良好的机器来有效地履行其职责，因为他们必须快速且连续地处理交易。

Starknet 当前的路线图包括去中心化 Sequencer 角色。这种向去中心化的转变将使更多的参与者成为测序者，从而为网络的稳健性做出贡献。

3.3 Provers （证明者）

证明者充当 Starknet 网络中的第二道验证线。他们的主要任务是验证排序器的工作（当它们收到排序器生成的块时）并生成这些过程已正确执行的证据。

证明者的职责包括：

1.接收区块：证明者从排序器获取已处理交易的区块。

2.处理：证明者第二次处理这些块，确保块内的所有交易都得到正确处理。

3.证明生成：处理后，证明者生成正确交易处理的证明。

4.将证明发送到以太坊：最后，将证明发送到以太坊网络进行验证。如果证明正确，以太坊网络就会接受该交易块。

证明者需要比测序者更多的计算能力，因为他们必须计算和生成证明，这是一个计算量很大的过程。然而，证明者的工作可以分为多个部分，从而实现并行性和高效的证明生成。证明生成过程是异步的，这意味着它不必立即或实时发生。这种灵活性允许将工作负载分配给多个证明者。每个证明者都可以在不同的块上工作，从而实现并行性和高效的证明生成。

Starknet 的设计依赖于这两种类型的参与者（排序器和证明器）协同工作，以确保交易的高效处理和安全验证。

3.4 Nodes (节点)

3.4.1

当谈到定义节点在比特币或以太坊中的作用时，人们经常将其角色误解为跟踪网络中的每笔交易。然而，这并不完全准确。

节点充当网络的审计员，维护网络的状态，例如每个参与者拥有多少比特币或特定智能合约的当前状态。他们通过处理交易并保存所有交易的记录来实现这一目标，但这是达到目的的手段，而不是目的本身。

在 Validity rollups 中，特别是在 Starknet 中，这个概念有些相反。节点不一定必须处理事务才能获取状态。与以太坊或比特币相比，Starknet 节点不需要处理所有交易来维护网络状态。

访问网络状态数据主要有两种方式：通过API网关或使用RPC协议与节点通信。操作您自己的节点通常比使用共享架构（例如网关）更快。随着时间的推移，Starknet 计划弃用 API 并将其替换为 JSON RPC 标准，从而使操作自己的节点变得更加有利。

值得注意的是，鼓励更多人运行节点可以提高网络的弹性并防止服务器泛洪，这一直是其他 L2 网络中的一个问题。

目前，节点主要有三种方法来跟踪网络状态，我们可以让节点实现这些方法中的任何一种：

1.重放旧交易：像以太坊或比特币一样，节点可以获取所有交易并重新执行它们。尽管这种方法很准确，但它不可扩展，除非您拥有一台能够处理负载的强大机器。如果您可以重放所有交易，您就可以成为排序者。

2.依赖 L2 共识：节点可以信任定序器来正确执行网络。当定序器更新状态并添加新块时，节点会认为更新是准确的。

3.检查 L1 上的证明验证：节点可以通过观察 L1 并确保每次发送证明时都收到更新的状态来监控网络状态。这样，他们就不必信任任何人，只需要跟踪 Starknet 的最新有效交易即可。

每种类型的节点设置都有自己的一组硬件要求和信任假设。

3.4.2 重放交易的节点

重放交易的节点需要强大的机器来跟踪和执行所有交易。这些节点没有信任假设；它们仅依赖于它们执行的交易，保证任何给定点的状态都是有效的。

3.4.3 依赖 L2 共识的节点

依赖 L2 共识的节点需要较少的计算能力。他们需要足够的存储来保存状态，但不需要处理大量交易。这里的权衡是信任假设。目前，Starknet 围绕一个 Sequencer 运行，因此这些节点信任 Starkware 不会破坏网络。然而，一旦 Sequencer 之间建立了共识机制和领导者选举，这些节点只需要相信一个 Sequencer 投入其股份来生成一个区块，并不愿意失去它。

3.4.4 在 L1 上检查证明验证的节点

仅根据 L1 上的证明验证更新其状态的节点需要最少的硬件。它们与以太坊节点具有相同的要求，一旦以太坊轻节点成为现实，维护这样的节点可能就像使用智能手机一样简单。唯一的权衡是延迟。证明并不是每个区块都会间歇性地发送到以太坊，从而导致状态更新延迟。计划更频繁地生成证明，即使它们没有立即发送到以太坊，从而允许这些节点减少延迟。然而，这一发展在 Starknet 路线图中还有很长的路要走。

3.5 智能合约

Starknet合约，是用cairo编写的程序，可以在starknet虚拟机上运行，它们可以访问starknet状态，并且可以与其他合约交互。

3.5.1 示例

存钱罐合约 (piggy bank contract) 是一种工厂合约模式，允许用户创建自己的个性化储蓄合约。在创建时，用户将指定他们的储蓄目标，可以是特定时间或特定金额，然后根据他们的储蓄目标创建并个性化子合同。

工厂合约密切关注创建的所有子合约，并将用户映射到他的个性化合约。用户在创建个性化储蓄合同后，可以按照他的目标进行存款和储蓄。但如果由于任何原因，用户在达到储蓄目标之前必须退出储蓄合同，则用户将支付提款金额 10% 的罚款。

该合约使用功能和所有权组件的组合来跟踪和维护上述功能。每个修改合约状态的函数调用也会发出事件。因此，充分理解这个合约示例的逻辑和实现将使您掌握开罗的组件系统、工厂标准模型、发出事件以及在 starknet 上编写智能合约时有用的许多其他方法。

3.5.2 存钱罐合约子合约

```
use starknet::ContractAddress;

#[derive(Drop, Serde, starknet::Store)]
enum target {
    blockTime: u128,
    amount: u128,
}

#[starknet::interface]
trait IERC20<TContractState> {
    fn name(self: @TContractState) -> felt252;
    fn symbol(self: @TContractState) -> felt252;
    fn decimals(self: @TContractState) -> u8;
    fn total_supply(self: @TContractState) -> u256;
    fn balanceOf(self: @TContractState, account: ContractAddress) -> u256;
    fn allowance(self: @TContractState, owner: ContractAddress, spender: ContractAddress) -> u256;
    fn transfer(ref self: TContractState, recipient: ContractAddress, amount: u256) -> bool;
    fn transferFrom(
        ref self: TContractState, sender: ContractAddress, recipient: ContractAddress, amount: u25
    ) -> bool;
    fn approve(ref self: TContractState, spender: ContractAddress, amount: u256) -> bool;
}

#[starknet::interface]
trait piggyBankTrait<TContractState> {
    fn deposit(ref self: TContractState, _amount: u128);
    fn withdraw(ref self: TContractState, _amount: u128);
    fn get_balance(self: @TContractState) -> u128;
    fn get_Target(self: @TContractState) -> (u128 , piggyBank::targetOption) ;
    // fn get_owner(self: @TContractState) -> ContractAddress;
    fn viewTarget(self: @TContractState) -> target;
}

#[starknet::contract]
mod piggyBank {
    use core::option::OptionTrait;
    use core::traits::TryInto;
    use starknet::{get_caller_address, ContractAddress, get_contract_address, Zeroable, get_block_
    use super::{IERC20Dispatcher, IERC20DispatcherTrait, target};
```

```
use core::traits::Into;
use piggy_bank::ownership_component::ownable_component;
component!(path: ownable_component, storage: ownable, event: OwnableEvent);


#[abi(embed_v0)]
impl OwnableImpl = ownable_component::Ownable<ContractState>;
impl OwnableInternalImpl = ownable_component::InternalImpl<ContractState>;

#[storage]
struct Storage {
    token: IERC20Dispatcher,
    manager: ContractAddress,
    balance: u128,
    withdrawalCondition: target,
    #[substorage(v0)]
    ownable: ownable_component::Storage
}

#[derive(Drop, Serde)]
enum targetOption {
    targetTime,
    targetAmount,
}

#[event]
#[derive(Drop, starknet::Event)]
enum Event {
    Deposit: Deposit,
    Withdraw: Withdraw,
    PaidProcessingFee: PaidProcessingFee,
    OwnableEvent: ownable_component::Event
}

#[derive(Drop, starknet::Event)]
struct Deposit {
    #[key]
    from: ContractAddress,
    #[key]
```

```
        Amount: u128,
    }

    #[derive(Drop, starknet::Event)]
    struct Withdraw {
        #[key]
        to: ContractAddress,
        #[key]
        Amount: u128,
        #[key]
        ActualAmount: u128,
    }

    #[derive(Drop, starknet::Event)]
    struct PaidProcessingFee {
        #[key]
        from: ContractAddress,
        #[key]
        Amount: u128,
    }

    mod Errors {
        const Address_Zero_Owner: felt252 = 'Invalid owner';
        const Address_Zero_Token: felt252 = 'Invalid Token';
        const UnAuthorized_Caller: felt252 = 'UnAuthorized caller';
        const Insufficient_Balance: felt252 = 'Insufficient balance';
    }

    #[constructor]
    fn constructor(ref self: ContractState, _owner: ContractAddress, _token: ContractAddress, _man
        assert(!_owner.is_zero(), Errors::Address_Zero_Owner);
        assert(!_token.is_zero(), Errors::Address_Zero_Token);
        self.ownable.owner.write(_owner);
        self.token.write(super::IERC20Dispatcher{contract_address: _token});
        self.manager.write(_manager);
        match target {
            targetOption::targetTime => self.withdrawalCondition.write(target::blockTime(targetDet
            targetOption::targetAmount => self.withdrawalCondition.write(target::amount(targetDeta
        }
```

```
}

#[external(v0)]
impl piggyBankImpl of super::piggyBankTrait<ContractState> {
    fn deposit(ref self: ContractState, _amount: u128) {
        let (caller, this, currentBalance) = self.getImportantAddresses();
        self.balance.write(currentBalance + _amount);

        self.token.read().transferFrom(caller, this, _amount.into());

        self.emit(Deposit { from: caller, Amount: _amount});
    }

    fn withdraw(ref self: ContractState, _amount: u128) {
        self.ownable.assert_only_owner();
        let (caller, this, currentBalance) = self.getImportantAddresses();
        assert(self.balance.read() >= _amount, Errors::Insufficient_Balance);

        let mut new_amount: u128 = 0;
        match self.withdrawalCondition.read() {
            target::blockTime(x) => new_amount = self.verifyBlockTime(x, _amount),
            target::amount(x) => new_amount = self.verifyTargetAmount(x, _amount),
        };

        self.balance.write(currentBalance - _amount);
        self.token.read().transfer(caller, new_amount.into());

        self.emit(Withdraw { to: caller, Amount: _amount, ActualAmount: new_amount});
    }

    fn get_balance(self: @ContractState) -> u128 {
        self.balance.read()
    }

    fn get_Target(self: @ContractState) -> (u128 , targetOption) {
        let condition = self.withdrawalCondition.read();
        match condition {
            target::blockTime(x) => {return (x, targetOption::targetTime);},
            target::amount(x) => {return (x, targetOption::targetAmount);},
```

```
        }
    }

    fn viewTarget(self: @ContractState) -> target {
        self.withdrawalCondition.read()
    }

}

#[generate_trait]
impl Private of PrivateTrait {
    fn verifyBlockTime(ref self: ContractState, blockTime: u128, withdrawalAmount: u128) -> u1
        if (blockTime <= get_block_timestamp().into()) {
            return withdrawalAmount;
        } else {
            return self.processWithdrawalFee(withdrawalAmount);
        }
    }

    fn verifyTargetAmount(ref self: ContractState, targetAmount: u128, withdrawalAmount: u128)
        if (self.balance.read() < targetAmount) {
            return self.processWithdrawalFee(withdrawalAmount);
        } else {
            return withdrawalAmount;
        }
    }

    fn processWithdrawalFee(ref self: ContractState, withdrawalAmount: u128) -> u128 {
        let withdrawalCharge: u128 = ((withdrawalAmount * 10) / 100);
        self.balance.write(self.balance.read() - withdrawalCharge);
        self.token.read().transfer(self.manager.read(), withdrawalCharge.into());
        self.emit(PaidProcessingFee{from: get_caller_address(), Amount: withdrawalCharge});
        return withdrawalAmount - withdrawalCharge;
    }

    fn getImportantAddresses(self: @ContractState) -> (ContractAddress, ContractAddress, u128)
        let caller: ContractAddress = get_caller_address();
        let this: ContractAddress = get_contract_address();
        let currentBalance: u128 = self.balance.read();
```

```
            (caller, this, currentBalance)
        }
    }
}
```

3.5.3 存钱罐工厂合约

```
use starknet::{ContractAddress, ClassHash};
use piggy_bank::piggy_bank::piggyBank::targetOption;
use array::ArrayTrait;

#[starknet::interface]
trait IPiggyBankFactory<TContractState> {
    fn createPiggyBank(ref self: TContractState, savingsTarget: targetOption, targetDetails: u128)
    fn updatePiggyBankHash(ref self: TContractState, newClasHash: ClassHash);
    fn getAllPiggyBank(self: @TContractState) -> Array<ContractAddress>;
    fn getPiggyBanksNumber(self: @TContractState) -> u128;
    fn getPiggyBankAddr(self: @TContractState, userAddress: ContractAddress) -> ContractAddress;
    fn get_owner(self: @TContractState) -> ContractAddress;
    fn get_childClassHash(self: @TContractState) -> ClassHash;
}

#[starknet::contract]
mod piggyFactory{
    use core::starknet::event::EventEmitter;
use piggy_bank::ownership_component::IOwnable;
    use core::serde::Serde;
    use starknet::{ContractAddress, ClassHash, get_caller_address, Zeroable};
    use starknet::syscalls::deploy_syscall;
    use dict::Felt252DictTrait;
    use super::targetOption;
    use piggy_bank::ownership_component::ownable_component;
    component!(path: ownable_component, storage: ownable, event: OwnableEvent);

    #[abi(embed_v0)]
    impl OwnableImpl = ownable_component::Ownable<ContractState>;
    impl OwnableInternalImpl = ownable_component::InternalImpl<ContractState>;

    #[storage]
    struct Storage {
        piggyBankHash: ClassHash,
        totalPiggyBanksNo: u128,
        AllBanksRecords: LegacyMap<u128, ContractAddress>,
        piggyBankOwner: LegacyMap::<ContractAddress, ContractAddress>,
        TokenAddr: ContractAddress,
        #[substorage(v0)]
```

```
    ownable: ownable_component::Storage
}

#[event]
#[derive(Drop, starknet::Event)]
enum Event {
    BankCreated: BankCreated,
    HashUpdated: HashUpdated,
    OwnableEvent: ownable_component::Event
}

#[derive(Drop, starknet::Event)]
struct BankCreated {
    #[key]
    for: ContractAddress,
}

#[derive(Drop, starknet::Event)]
struct HashUpdated {
    #[key]
    by: ContractAddress,
    #[key]
    oldHash: ClassHash,
    #[key]
    newHash: ClassHash,
}

mod Errors {
    const Address_Zero_Owner: felt252 = 'Invalid owner';
}

#[constructor]
fn constructor(ref self: ContractState, piggyBankClassHash: ClassHash, tokenAddr: ContractAddr
    self.piggyBankHash.write(piggyBankClassHash);
    self.ownable.owner.write(_owner);
    self.TokenAddr.write(tokenAddr);
}

#[external(v0)]
```

```
impl piggyFactoryImpl of super::IPiggyBankFactory<ContractState> {
    fn createPiggyBank(ref self: ContractState, savingsTarget: targetOption, targetDetails: u1
        // Contructor arguments
        let mut constructor_calldata = ArrayTrait::new();
        get_caller_address().serialize(ref constructor_calldata);
        self.TokenAddr.read().serialize(ref constructor_calldata);
        self.ownable.owner().serialize(ref constructor_calldata);
        savingsTarget.serialize(ref constructor_calldata);
        targetDetails.serialize(ref constructor_calldata);

        // Contract deployment
        let (deployed_address, _) = deploy_syscall(
            self.piggyBankHash.read(), 0, constructor_calldata.span(), false
        )
            .expect('failed to deploy counter');
        self.totalPiggyBanksNo.write(self.totalPiggyBanksNo.read() + 1);
        self.AllBanksRecords.write(self.totalPiggyBanksNo.read(), deployed_address);
        self.piggyBankOwner.write(get_caller_address(), deployed_address);
        self.emit(BankCreated{for: get_caller_address()});

        deployed_address
    }

    fn updatePiggyBankHash(ref self: ContractState, newClasHash: ClassHash) {
        self.ownable.assert_only_owner();
        self.piggyBankHash.write(newClasHash);
        self.emit(HashUpdated{by: self.ownable.owner(), oldHash: self.piggyBankHash.read(), ne
    }

    fn getAllPiggyBank(self: @ContractState) -> Array<ContractAddress> {
        let mut piggyBanksAddress = ArrayTrait::new();
        let mut i: u128  = 1;
        loop {
            if i > self.totalPiggyBanksNo.read() {
                break;
            }
            piggyBanksAddress.append(self.AllBanksRecords.read(i));
            i += 1;
        };
```

```
            piggyBanksAddress
        }

        fn getPiggyBanksNumber(self: @ContractState) -> u128 {
            self.totalPiggyBanksNo.read()
        }
        fn getPiggyBankAddr(self: @ContractState, userAddress: ContractAddress) -> ContractAddress
            assert(!userAddress.is_zero(), Errors::Address_Zero_Owner);
            self.piggyBankOwner.read(userAddress)
        }
        fn get_owner(self: @ContractState) -> ContractAddress {
            self.ownable.owner()
        }

        fn get_childClassHash(self: @ContractState) -> ClassHash {
            self.piggyBankHash.read()
        }

    }

}
```

4 Account Abstraction (账户抽象)

4.1 简介

账户抽象（AA）代表了一种在区块链网络中管理账户和交易的方法。它涉及两个关键概念：

1.交易灵活性：

- 智能合约验证其交易，摆脱通用验证模型。
- 好处包括涵盖汽油费的智能合约、支持一个账户的多个签名者以及使用替代加密签名。

2.用户体验优化：

- AA 使开发人员能够设计灵活的安全模型，例如对常规和高价值交易使用不同的密钥。
- 它提供了帐户恢复种子短语的替代方案，简化了用户体验。

从技术上讲，AA 用更广泛的账户概念取代了外部拥有账户 (EOA)。在这个模型中，账户是智能合约，每个合约都有其独特的规则和行为。这些规则可以管理交易排序、签名、访问控制等，提供广泛的定制。

AA的主要定义：

定义 1：正如 Martin Triay 在 Devcon 6 上所描述的，AA 允许智能合约为其交易支付费用。这与传统的外部账户或智能钱包不同。

定义 2：Devcon 6 上的 Lightclient 将 AA 定义为验证抽象。与以太坊的第 1 层单一验证方法不同，AA 允许各种签名类型、加密方法和执行流程。

4.2 账户抽象的应用

4.2.1 账户抽象（AA）增强了区块链技术中自我托管的可访问性和安全性。

以下是 AA 启用的一些功能：

硬件签名者：

AA 支持使用存储在智能手机安全飞地中的密钥进行交易签名，并结合生物识别身份以增强安全性和易用性。

恢复：

如果密钥丢失或受损，AA 可以安全地替换密钥，从而无需种子短语并简化用户体验。

按键轮换：

如果密钥被泄露，可以轻松更换，无需转移资产。

会话键：

AA 促进了 web3 应用程序的一次登录功能，允许代表您进行交易并最大限度地减少持续批准。

自定义交易验证方案：

AA支持各种签名方案和安全规则，可以根据个人需求定制安全措施。

4.2.2 AA 还通过多种方式增强安全性：

改进的密钥管理：

多台设备可以链接到您的钱包，即使一台设备丢失也能确保帐户访问。

多样化的签名和验证方案：

AA 提供额外的安全措施，例如针对大量交易的双因素身份验证，以满足个人安全需求。

自定义安全策略：

可以针对不同的用户类型或设备定制安全性，结合银行和 Web2 行业的最佳实践。

4.3 Ethereum Account System (以太坊账户系统)

了解以太坊的活期账户系统对于理解账户抽象（AA）的好处非常重要。以太坊的账户系统包括两种类型：

4.3.1 外部拥有账户 (EOA)：

由以太坊网络外部的个人、钱包或实体使用。

由从加密签名者的公钥派生的地址进行标识，其中包括私钥和公钥。

私钥对交易或消息进行签名以证明所有权，而公钥则验证签名。

交易必须由 EOA 的私钥签名才能修改账户状态，通过唯一的加密身份确保安全。

4.3.2 合约账户 (CA)：

本质上是以太坊区块链上的智能合约。

缺乏私钥并由来自 EOA 的交易或消息激活。

他们的行为是由他们的代码定义的。

4.3.3 账户模型经常面临的挑战包括：

密钥管理：

丢失私钥意味着账户控制权和资产的不可逆转的损失。

如果被盗，窃贼就可以完全访问该帐户及其资产。

用户体验：

以太坊帐户模型目前缺乏用户友好的密钥或帐户恢复选项。

加密钱包等复杂的界面可能会阻止非技术用户，从而限制更广泛的采用。

缺乏灵活性：

传统模型限制了自定义交易验证方案，限制了潜在的安全性和访问控制增强。

AA 旨在解决这些问题，提供改进安全性、可扩展性和用户体验的机会。

4.4 为什么以太坊第一层还没有实现账户抽象？

以太坊的第 1 层 (L1) 目前在协议级别缺乏对账户抽象 (AA) 的支持，这并不是因为对其价值缺乏兴趣或认识，而是因为其集成所涉及的复杂性。

在以太坊 L1 中实施 AA 的主要挑战包括：

4.4.1外部拥有账户 (EOA) 的根深蒂固性质：

EOA 是以太坊核心协议不可或缺的一部分。

修改它们以支持 AA 是一项艰巨的任务，特别是随着以太坊的价值和使用量不断增长。

4.4.2 以太坊虚拟机（EVM）的局限性：

EVM（以太坊的智能合约运行时环境）面临着阻碍 AA 实施的限制。

尽管自以太坊成立以来提出了多项 AA 提案，但由于其他关键更新和改进的优先顺序，这些提案已被推迟。

4.4.3 Layer2 解决方案：

L2 解决方案注重可扩展性和性能，更适合 AA。

Starknet 和 ZKSync 等平台受到 EIP4337 的启发，是原生 AA 实现的先驱。

由于将 AA 集成到以太坊 L1 的持续延迟和复杂性，许多拥护者已经转移了他们的焦点：

4.4.4 转向第 2 层倡导：

支持者现在支持通过 L2 解决方案采用 AA，而不是等待 EOA 逐步淘汰以及 AA 融入以太坊核心。

这种方法旨在更快地为用户提供 AA 收益，并保持以太坊在快速发展的加密货币领域的竞争优势。