**填表说明**

1．只有学籍状态为注册或暂缓注册的研究生才允许开题。但学籍状态为暂缓注册的研究生只有在完成注册手续之后，开题报告及其评审结果才能被认可。

2．开题报告为A4大小，于左侧装订成册。硕士研究生应逐项认真填写，各栏空格不够时请自行加页。

3．开题报告经指导教师审阅通过后，由硕士研究生公开宣读，并接受专家组质疑、评议。考核小组由5名及以上本专业学位类别（领域）的副高级及以上职称专家或硕士生导师组成，导师可为考核小组成员，但不得担任考核小组组长。评审合格后，由研究生装订后交指导教师留存备查，同时把含签字的完整开题报告电子版上传研究生管理系统。

4．硕士研究生应在选题前阅读至少30篇本研究领域及方向的国内外文献，了解、学习本领域的前沿和研究进展，并写出不少于3000字的文献综述报告。文献综述报告应反映国际和国内本领域的研究历史、现状和发展趋势。文献综述报告是开题报告的必要附件，开题报告通过后，由研究生装订后交指导教师留存备查，同时把含签字和评阅意见的完整文献综述电子版上传研究生管理系统。

5．“参考文献”著录按照GB7714-87文参考文献著录规则执行。书写顺序为：序号·作者·论文名或著作名·杂志或会议名·卷号、期号或会议地点·出版社·页号·年。

6. 本件只接受非涉密内容

**一 选题依据**

|  |
| --- |
| 简述该选题的研究意义、国内外研究概况和发展趋势。 |
| 选题背景及意义  近年来，越来越多的开发者为rust编程语言的高内存安全性和高效率性所吸引，愿意使用之以开发自己的操作系统及配套的软件生态环境。然而，操作系统开发的关键点之一在于丰富的软件生态环境，这正是新兴操作系统所欠缺的。鉴于Linux操作系统的生态环境相对成熟性，将Linux应用程序移植到生态环境尚不成熟的的新兴操作系统中成为了快速拓展新兴操作系统软件生态环境的方案之一。移植软件，首先需要理解移植对象的业务逻辑。然而，各种已有软件中广泛存在的代码异味，尤其是死代码，使得理解它们的业务逻辑变得更加困难。  目前，已有不少工作针对部分主流编程语言提出了死代码移除的技术方案。然而，作为新兴系统级编程语言的rust编程语言却缺少类似的技术方案。目前，rust语言已被广泛应用于操作系统、嵌入式设备等领域的开发工作。这种工作代码量巨大，且需要进行反复的迭代和尝试，从而更容易累积死代码。然而，无论是从代码质量和可维护性的角度出发，还是从嵌入式设备上装载的、有限的存储空间的角度出发，死代码移除之于rust软件项目的重要性只增不减。  本选题提出一种基于函数调用图进行死代码探测的技术方案，帮助开发者更高效地移除死代码，在简化软件业务逻辑理解的同时，降低代码库体积，提升软件运行效率；为验证这一方案的可行性，本选题还将以实际rust软件项目为对象进行死代码移除，并移植到真实操作系统中，使之成为顺应Web 3.0时代潮流的区块链操作系统。  国内外研究现状  死代码的概念  死代码是一种广泛存在于软件源代码中的代码异味。不同学派对死代码的定义不尽相同。Brown等人将死代码定义为在不断变化的软件设计中始终未移除的未使用代码[BrownBook] 。Mantyla等人认为：死代码就是过去使用过，但目前已不会再被执行的源代码[Mantyla2003] 。Wake将未使用的变量、函数参数、类成员属性、类方法和类本身视为死代码[WakeBook] 。Martin将死代码定义为从未执行过的代码（例如永假if内的语句块），而死函数是永远不会被调用的方法[MartinBook] 。而在程序设计语言领域，死代码是指其结果从未被使用的计算（例如，在代码中引用的变量，但在运行时实际上不使用）[CodeCompaction] 。虽定义不同，其本质并无二样，即死代码是在程序运行过程中永不可能被执行的部分。  虽然死代码并不会被执行，但它们对软件开发与维护仍具有负面影响，主要可以归结为以下几点[MultiStudy]：  • 令代码更难理解: 开发者更难理解代码的结构和用意[BetterUnderstanding][MetricsUnderstanding] 。对于经验尚不丰富的新开发人员，他们可能会误以为死代码是有用的，从而轻则花费不必要的时间弄懂死代码、重则任其累积令代码库的质量越来越差。  • 令代码更难维护：让维护工作变得更加复杂，在日常维护或升级迭代时影响开发人员的工作效率、在降低代码质量的同时可能还会引入新的缺陷甚至错误[SoftwareAgeing] 。  • 徒增开发工时：开发者花费无用的时间维护死代码或对其进行debug，而这部分工作对项目并无任何帮助。  • 降低运行效率：虽非总是如此，但死代码有可能降低软件运行效率或徒增内存占用。  死代码探测与移除技术  目前，针对不同编程语言的死代码探测及移除技术不断涌现，其中较具代表性的有JavaScript的Tree Shaking技术方案，PHP的Web系统的动态标记技术方案和Java的DUM技术方案。  Tree Shaking技术方案依赖ES2015标准引入的import、export模块语法进行死代码移除。它通过分析依赖关系，确定未使用的模块并在最终生成的代码中将它们剔除。然而其简单的技术原理与思想也带来了局限性：它只能以模块为单位，而不能以模块中具体的功能为单位进行移除，导致生成的源代码体积仍然过大；另外，该过程对开发者透明，开发者无从知晓哪些模块被删除，因此无法辅助开发者主动提高代码质量。  PHP的Web系统动态标记技术方案[PHPWebSystem] 首先收集Web系统用到的所有文件，然后为其标注元数据，包含首次、最后使用时间，使用次数等。然后运行该Web系统，利用动态分析技术维护并追踪上述指标的变化[DynAnalysis] 。运行系统一段时间以后查阅元数据，即可得知哪个文件是冗余或不常用的。该方案在Hostnet工业规模中进行了测试，并安全高效地移除了30%的原始代码库中的死代码。虽然如此，动态分析技术的特性决定了运行该技术方案需要覆盖率足够广、持续时间足够长的测试以减少漏检；且最终结果需要人工决策，无法实现完全自动化。同时，它也和Tree Shaking一样，只能实现文件级的，粒度较大的死代码移除。  Java的DUM技术方案提出了一种基于静态分析的技术方案，用于探测Java桌面软件中的不可达方法[DUM] 。它被设计为在Java字节码上工作，利用其中的信息将源程序转换为有向图表示[GraphBuilding] 。建图完成后，通过从一个起始节点开始遍历之来识别可达节点，其余即视为不可达节点（代表不可达方法）。与JTombstone、Google CodePro AnalytiX等行业已有工具比较，DUM表现出了更高的查准率。与前两种技术方案相比，DUM的分析粒度可达方法级别，远远细于Tree Shaking的模块级别和PHP动态标记技术的文件级别，使之能够移除尽可能多的死代码，最大地减轻开发者理解已有软件业务逻辑的心智负担，提升软件代码库质量及其运行效率。  综上所述，死代码对软件项目的负面影响使其成为有必要移除的代码异味，目前较有代表性的技术方案均有其优越性及局限性，在这其中，基于函数调用图的静态分析方法能够实现细粒度的死代码探测，其良好的效果与函数调用图在主流编程语言上的广泛性使得它成为在rust编程语言上实现死代码探测的理想途径。  CG-RTL函数调用图生成工具  编译型程序设计语言的编译工具链在将源代码编译为目标平台的二进制可执行文件时，往往会选择先将源代码编译为某种形式的中间表示（IR），再将中间表示编译为目标平台的机器代码。文献 [cg\_rtl] 提出一种基于寄存器传送语言（Register Transfer Language，RTL）中间表示的分析方法，从GCC编译器输出的RTL中间表示中，利用字符串处理提取当前软件包中的函数定义、函数调用信息，与其他软件包的上述信息进行整合，最终绘制成一张函数调用有向图。在此基础上，研究团队又提出了能够处理动态函数调用的DCG-RTL [dcg\_rtl] 和基于数据库的函数调用图生成工具 [dbcg\_rtl] 。  CG-RTL解决了已有工具需要基础知识、产生过量冗余信息、和编译环境耦合度过高的问题。虽其仅适用于C语言的RTL中间表示分析，但其利用中间表示进行逐模块分析的方法仍然值得借鉴。  rust的中间表示  在部分语言中，为方便从不同的角度进行全面完善的检查，一些语言使用数种不同层级的中间表示，待上层级中间表示通过检查后，再将其编译为下一层级的中间表示，运行下一步的检查，如此逐层降低层级（Lowering），直至获得最后的机器码。这些中间表示从不同角度呈现了不同的信息，有利于软件开发者开发外部工具对其进行诸如静态检查等操作。rust程序设计语言亦采用了这种多层级中间表示的思路。按层级由高到低，rust一共使用了如下四种中间表示：  1. 高层级中间表示（HIR）：HIR是rust中最高层级的中间表示，由对源代码进行语法解析、宏展开等处理之后的抽象语法树转换而来。其形式和rust源代码尚有相似之处，但将一些语法糖展开为了更易于分析的形式，例如for循环将被展开为loop循环等。但由于此时rustc编译器尚未进行类型检查，因此HIR中的类型信息较为模糊，不适合作为静态分析工具的输入。  2. 带类型的高层级中间表示（THIR）：该中间表示是由HIR在完成类型检查后降低层级而来，主要用于枚举穷尽性检查、不安全行为检查和下一层中间表示的构造。和HIR相比，THIR最大的不同在于诸如struct和trait等结构将不会在THIR中出现，因为THIR仅保留了源代码中可执行的部分，例如定义的普通函数以及impl块中定义的关联函数、方法等。由于具有上述“仅保留可执行部分”以及结构比HIR更简洁的特点，THIR非常适合用于分析rust crate中的函数定义信息。  3. 中层级中间表示（MIR）：这种中间表示于RFC 1211中初次引入 ，用于控制流相关的安全检查，例如借用检查器。它进一步将一些语法糖展开，引入了在rust源代码中不可能出现的语句，同时也会执行控制流分析。由于rustc提供了一组不稳定的API接口用于和MIR交互，MIR成为了诸多外部工具处理rust程序代码的不二选择，如MIRChecker [MirChecker] 、Kani [Kani] 等均采用MIR作为其分析对象。  Prazi与函数调用依赖关系网络  随着托管平台投毒等安全威胁的出现，基于包依赖关系网络（Package Dependency Network，PDN）的大粒度分析已不能满足当下软件安全分析的需求。因此，Joseph Hejderup等人提出了一种全新的依赖关系网络：函数调用依赖关系网络（Call-graph Dependency Network，CDN） [Prazi] ，并利用之进行更细粒度的分析。为获得函数调用依赖关系网络，他们开发了Prazi分析器。这是一款基于MIR中间表示、利用Docker的虚拟环境进行软件包编译、借助rust-callgraphs工具生成函数调用图的分析工具，能够针对一个crate生成它的函数调用依赖关系网络。  为证明该方案的可用性，作者团队为crates.io上的所有crate都使用prazi进行了分析，并通过分析统计数据得出了有价值的结论。在crates.io托管的所有软件包中，50%的函数调用是在调用外部依赖项中的函数。不仅如此，虽然一个crate在其78.8%的直接依赖项中至少会调用一个函数，但在其传递依赖项中至少调用一个函数的概率却锐减至40%，这表明软件包的所有传递依赖项中有一半以上可能没有被调用。  虽然受制于rust编译器提供的MIR编程接口的不稳定性，Prazi分析器已无法使用，但其证明了在rust项目上生成函数调用图并进一步分析的技术可行性；同时，传递依赖项的函数调用率锐减也反映了rust软件项目中死代码存在的广泛性，进一步证明了选题的意义。  MIRAI与Rupta  MIRAI由来自Facebook的技术团队开发，是一款工作在rust编程语言的中层中间表示（MIR）上的静态分析工具 。与追求创新性的学术用途原型工具不同，MIRAI追求在实际工业生产环境中的实用性。其宗旨是在尽可能低的假阳率下定位存在异味的rust代码，并给出切实可行的修改建议。  MIRAI的功能之一是函数调用关系生成功能（下简称为MIRAI-CGG）。MIRAI-CGG支持为指定crate生成函数调用关系信息，这些信息不仅可以输出为.dot文件，供Graphviz工具绘制为矢量图，还能输出JSON格式的函数调用位置（callsite）信息，包含发生调用的源代码文件虚拟路径、行号、列号，以及调用者（called）与被调用者（callee）的信息，十分全面。然而，其低假阳率的目标并不完全适用于分析死代码。经过试验发现，MIRAI对函数调用情况存在漏报的情况，开发者若完全按照MIRAI的输出进行死代码移除，有概率将具有实际作用的代码移除导致软件功能缺失甚至编译不通过。因此，有必要对MIRAI进行必要的修改，降低其漏报率，方能用于死代码移除。  Rupta[Rupta] 是一款上下文敏感指针分析框架，观察其代码库易知其脱胎于MIRAI而优于MIRAI，在构建函数调用图（call graph）方面的能力优于MIRAI。Rupta 采用基于调用点的上下文敏感性算法，为Rust中常见的静态分派、动态分派、嵌套数据结构等情况分别采用了针对性的算法方案，从而提供更精确的分析结果。与现有的两种技术 Rurta（基于快速类型分析）和 Ruscg（仅静态分派）相比，Rupta 在完整性上发现更多的调用关系，并在精度上消除了大约 70% 的虚假动态调用关系。  将MIRAI与Rupta比较，不难得出其各自的优越性：MIRAI输出的信息格式规整，利于分析；但其低假阳率的设计初衷导致调用关系不全，存在漏报的问题；Rupta的查全率显著更优，但其仅能输出.dot格式的有向图文件，缺少了分析死代码所需的必要信息。二者均需经过不同程度的修改，方能用于本选题的应用场景中。  软件移植研究现状  代码复用是提升软件产品开发效率的一种常用且有效的方法。常见的代码复用手段主要包含软件模块化设计和软件移植。前者将大型软件拆分成小的可复用的模块，供其他开发者选用并成为他们所开发的软件的一部分；后者将大型软件在不同的操作系统甚至不同的硬件平台上进行移植，以拓宽同样功能代码的泛用性。  软件移植技术是关心后者的，即将软件从一个操作系统或硬件平台移植到另一个的技术。于开发商而言，软件移植以小于重新开发的代价令已有软件在新平台上得以运行，有益于拓宽市场，优化用户使用体验。然而，跨操作系统甚至硬件平台的移植并非易事，因为开发者在这一过程中可能会遇到来自硬件和软件方方面面的阻碍，例如硬件上的大小端存储、软件上的目标平台的系统调用实现不全、编程语言本身存在未定义行为导致在不同编译器实现中表现出不同的行为等问题。因此，评估软件移植难度，并设法降低这个难度便成为了众多学者与开发者追求的共同目标 [DesirablePortability] 。  在Wolberg等人的研究 [PortingAndLoC] 中，代码库尺寸（即代码行数）与移植难度呈现指数级正相关。文献 [HAKUTA1997145] 在前者的基础上进一步将阻碍移植因素、移植代价因素纳入考量，获得了一个评估量化移植难度的模型。在构建模型时，作者团队发现移植效率的差异本质上取决于移植工程师的经验和技巧的差异，以及开发和测试环境的差异。因此，他们也呼吁从移植辅助工具、移植指南手册和软件设计准则三方面入手降低移植工作的难度。  上述两项工作的共同结论之一是，代码行数的增加会令移植工作更具挑战性。这从侧面证明了本选题所采取技术手段的合理性：通过移除不必要的代码，减少代码行数，从而降低移植工作难度，提升移植效率。  Substrate区块链  Substrate由以太坊项目的联合创始人Gavin Wood率领Parity团队开发，是一个基于rust的开源区块链框架。它对于多链结构提供了较好的支持，能够克服传统单链区块链在一条链上保有大量用户、存储大量信息，导致运行效率降低的问题，且针对多链间通信提出了一套解决方案。不仅如此，高度可定制的灵活模块化设计模式，使得Substrate能够以功能模块（称之为pallet）为单位进行组件增删，令开发者得以将Substrate打造成最契合他们需求的区块链。  目前，官方已推出了便于开发者开发的Substrate Node Template（下简称为SNT） 。它完全基于rust编程语言开发，内含一个包含最基本功能的Substrate全节点实现。其代码库大小适中，既能体现本选题提出的死代码探测方案的泛用性，又不至于因代码量过大导致分析过分繁杂。此外，区块链技术与操作系统内核的融合领域的研究工作较少，本课题亦可借此机会探索区块链技术与操作系统融合的形式，探究此举对操作系统技术发展的意义。  SNT的代码结构  SNT的代码库如[img1] 所示，其本质是一个Rust工作空间（workspace），内含三个成员：node, runtime和pallets。这其中，pallets成员存储SNT使用的所有自定义功能模块（即上文中提及的pallet），内仅含一个模板pallet，称之为template；runtime成员主要定义了SNT在运行时的链上状态转换逻辑，为单lib.rs文件结构；node成员负责P2P网络通信、区块产生和确认（finalization）、处理外部RPC请求等链外事务，内含多个源代码文件，有继续细分的必要。  代码阅读的结果表明，node成员以8份rust源代码组成，其中：  • lib.rs负责将自身的rpc，chain\_spec与service模块暴露给外界使用，而main.rs仅负责启动command模块中的run函数。它们并未为SNT实现更多功能，在后续代码分析时可以忽略。  • cli模块定义了一系列子命令，而command模块则利用前者定义的子命令结合用户传入的命令行参数进行解析，并根据之采取不同的行为。二者联合为SNT提供了命令行参数解析的服务，可以在逻辑上合并为一个逻辑模块。  • chain\_spec模块实现了初始化链时的配置选项，例如链的命名与唯一标识，链中预置的账户信息和账户余额等等。  • rpc  • service  • benchmarking    Substrate Node Template代码结构  SNT的大致依赖关系如[img2] 所示。结合上文对各模块功能的描述，可给出一拓扑排序，指示令SNT自底向上摆脱具体操作系统依赖的实现顺序。    Substrate Node Template大致依赖关系  SNT与具体操作系统的耦合强度  显然地，软件移植的难度与其与某具体操作系统的耦合程度呈负相关关系。同时，同一个软件的不同组成部分对具体操作系统的依赖性亦有所不同。例如生成随机数、计算密钥对和定义声明宏等操作，它们几乎不依赖具体操作系统提供的服务，因而可以在诸如AsyncOS等由纯Rust实现的操作系统内核中作为依赖项直接引用。而诸如命令行参数解析、协程管理等操作需要操作系统提供的服务代为管理软硬件资源，因而无法直接在纯Rust实现的操作系统中运行。  为评估将SNT移植入内核的难易程度，有必要对SNT的一级依赖项与操作系统的耦合度进行评估。以AsyncOS为基底，逐一尝试将它们作为依赖项引入AsyncOS的Cargo清单文件中，并观察编译是否顺利结束，从而得出了以下结果：  • pallets/template组件：5个依赖项中的3个与AsyncOS兼容。  • runtime组件：29个依赖项中的7个与AsyncOS兼容。  • 组件：37个依赖项中的2个与AsyncOS兼容。  根据依赖项的命名规则不同，亦能发现一些规律，如下表所示：   |  |  | | --- | --- | | 依赖项命名前缀 | 与AsyncOS的兼容性情况 | | frame\_\* | 大多无法兼容 | | pallet\_\* | 完全无法兼容 | | sp\_\* | 大约有五成可兼容 | | sc\_\* | 完全无法兼容 |   为区分不同用途的依赖项，SNT采用了一种特殊的命名规则：以sc\_\*为前缀的依赖项负责节点间通讯，以frame\_\*或pallet\_\*为前缀的依赖项负责链上状态转换，以sp\_\*为前缀的依赖项负责以上二者的数据交换。在SNT的模块设计中，节点间通信模块是一切服务的基石，其中即有与具体操作系统无关的功能，也有诸如p2p网络等有依赖于操作系统提供的服务的功能；而其余两个功能模块依赖于节点通信模块，依赖层级均高于后者。于是，不难得出结论：随依赖层级逐渐上升，AsyncOS和SNT的兼容性越来越差（无法通过编译之依赖项的占比越来越大）。  虽然有诸多依赖项需要修改，但在测试过程中发现的一些现象亦值得注意。多数依赖项在编译报错时的错误信息具有不少共性，例如找不到Ok、Result等常见的Rust结构体等。这些错误信息表明，依赖项在编译时使用了rust-std，而AsyncOS操作系统没有提供这一库。只需将其替换为rust核心库（core）中具有相同功能的同名结构体，问题便可解决。  一言以蔽之，SNT的部分组件对AsyncOS环境已经具有兼容性，而剩余兼容性不佳的组件亦具备改造的可能。通过核心库中的rust常用数据结构对依赖项中使用的标准库中的数据结构进行替换，同时扩展AsyncOS的功能从而使得其能提供更多样的服务，将SNT移植入AsyncOS作为用户态程序运行，甚至内核态中的系统服务运行都是完全可能的。 |

**二 研究内容**

|  |
| --- |
| 论证提出课题研究的总体思路，拟开展的主要研究内容及内涵，说明各部分研究内容之间的相关关系。  本选题提出一种基于函数调用图进行死代码探测的技术方案，帮助开发者更高效地移除死代码；为验证这一方案的可行性，本选题还将以Substrate区块链项目为对象进行死代码移除，并移植到AsyncOS操作系统中。工作大致可分为三个阶段：  • 改进并利用Rupta分析工具，获取rust软件项目的函数调用图。采用基于函数调用图的方法探测死代码并标记之，以求在简化软件业务逻辑理解的同时，降低代码库体积，提升软件运行效率。  • 将经过代码裁剪的Substrate的P2P网络部分移植到AsyncOS操作系统中，为打造区块链操作系统打下基础。  • 基于移植完成的P2P网络机能，向AsyncOS中移植更多SNT的区块链功能特性，使之成为顺应Web 3.0时代潮流的操作系统内核。 |

**三 研究方案**

|  |
| --- |
| 拟采用的研究方法、技术路线、实验方案及可行性分析。  死代码探测实施方案  目前，市面上尚无针对rust编程语言进行死代码探测的工具。故本选题计划以Rupta为基底，以死代码探测为目标进行修改，去除不必要的功能并完善函数调用图生成功能，以构造基于函数调用图的死代码探测技术方案。为验证该方案的可行性，取Substrate全节点客户端作为待分析项目，探测并移除其死代码，获得精简的Substrate全节点客户端。  函数调用图的构造  所谓rust项目的函数调用图，乃是一张有向图。图中，每个节点均代表一个该项目中出现的函数；若函数A调用了函数B，则从代表函数A的节点引一条指向代表函数B的节点的有向边。通过遍历该有向图，能够区分可达节点和不可达节点，进而确定不可达函数，从而提示开发者保留可达函数，移除不可达函数，达到死代码移除的目的。  MIRAI和Rupta均工作在rust编程语言的MIR中间表示上，其最小分析粒度为函数。由rust中间表示的特性可知，自THIR开始，rustc就抛弃了诸如struct、impl块等内容，而将所有的方法均展开成为了普通的函数定义。在THIR及之后的中间表示中，函数定义始终以body的形式存在，且每个函数均以被称为DefId的编号唯一标识。这种行为非常符合选题构造函数调用图的需求。  对Substrate全节点客户端分别调用MIRAI和Rupta进行函数调用关系分析发现，Rupta输出的调用关系文件体积远大于MIRAI输出的关系文件，为后者的250倍。浏览其中的内容亦发现，Rupta发现了实际存在、且MIRAI未发现的函数调用关系。因此，本选题决定在基于MIRAI而优于MIRAI的Rupta上进行二次开发，令其能够输出更多、更结构化、更便于分析的信息，以供死代码探测之用。  正如前文所言，rustc为MIR提供了一组不稳定的API供开发者调用，其中提供的rustc\_driver::Callbacks特征（trait）允许开发者介入rustc编译rust项目的不同阶段，获取编译信息并执行自定义的回调函数进行分析等操作。  Rupta实现了该特征中的after\_analysis回调方法，该回调方法在rustc编译获得MIR之后，继续降层为LLVM IR之前调用，可获得编译器在将源代码编译到MIR中间表示中收集的所有信息。通过该方法，Rupta进行了自顶向下，由最大范围的rust包（crate）级分析到最小范围的函数调用级分析的过程。  结构化信息输出  现阶段，Rupta的输出存在函数所属Crate相关信息缺失导致死代码探测工作困难等问题。因此，需要通过在分析过程中修改代码，收集缺少的信息，同时重构或另外实现一套信息输出机制，以合理的、有利于标识死代码的结构展示分析结果。预期的分析结果条目大致如下：  {  // 待分析对象及其所依赖的所有crate  "crates": [  {  // crate的唯一标识  "crate\_name": ...,  // 该crate的Cargo.toml在文件系统中的路径  "manifest\_path": ...  }, ...  ],  // 待分析目标中的所有函数  "callables": [  {  // 所述的crate在crates数组中的下标  "belongs\_to\_crate\_idx": ...,  // 该函数在文件系统中的路径  "source\_file\_path": ...,  // 该行数定义所在的行号  "line\_number": ...  }, ...  ],  "calls": [  {  "caller\_idx": ...,  "callee\_idx": ...,  "line\_number": ...,  "file": ...  }, ...  ] }  整体内容和Rupta现阶段的分析结果输出相比结构和内容均有不同，主要增加了crate的完整名称及位置信息，方便开发者区分同一crate的不同版本，更有针对性地进行死代码探测及移除；同时，为每一个函数补充其所属的crate的标记，从而区分不同crate中的同名函数，实现更有针对性地进行死代码移除。 跨操作系统移植实施方案 考虑到AsyncOS操作系统同样以rust写成，且结构简单利于分析，故以AsyncOS为移植目标平台，对精简的Substrate全节点客户端开展移植工作。这包含替换或重写不适配AsyncOS运行环境的依赖项，并为AsyncOS实现必须的系统调用。最终，将AsyncOS打造成包含部分区块链特性的区块链操作系统。 rust-std标准库 为方便开发者在主流操作系统上进行开发，降低开发人员负担，rust为其在主流操作系统上的实现配备了标准库rust-std。下至内存分配、缓冲区管理，上至网络通信等功能，rust-std均提供了支持，封装了与操作系统交互的内容，使得开发者能够专注于开发软件业务逻辑上。AsyncOS作为新兴的教学操作系统，rust并未为其提供标准库实现，故想将应用移植到AsyncOS上，有两个必须进行的步骤：   1. 替换或改写需要标准库支持的依赖项 2. 实现应用需要的系统调用   随着嵌入式开发在rust社区中的兴起，在无标准库支持的设备上进行rust开发的需求水涨船高，一个名为no-std特性的概念也应运而生。若一个crate声称自己对no-std环境兼容，则该crate能够在无标准库支持的环境下提供至少一部分功能供应用使用。在诸如crates.io和lib.rs这些rust crate介绍平台上，支持此类特性的crate往往会为自己标注no-std的tag。在进行现有crate替换时，应尽可能采用这些支持no-std特性的crate，以降低改写工作量。 SNT改造计划 正如前文所述的兼容性测试所言，虽然SNT的部分依赖项无法在AsyncOS中直接使用，但可以通过使用rust核心库提供的服务，以及AsyncOS提供的（以及未来能够提供的）服务加以替代，从而实现区块链操作系统的实现。因此，根据依赖层级由低向高的顺序，进行自顶向下和自底向上结合的依赖项改写工作。考虑到p2p网络功能为区块链所有服务的基石，故计划从网络模块开始改写，令SNT使用的网络模块与AsyncOS已经提供的网络通信原语相兼容；再以此为基础，逐步地尝试实现更多区块链的功能。 |

**四 研究工作进度安排**

|  |
| --- |
| 理论研究：应包括文献调研，理论推导，数值计算，理论分析，撰写论文等；实验研究和工程技术研究：应包括文献调研，理论分析，实验设计，仪器设备的研制和调试，实验操作，实验数据的分析处理，撰写论文等。   * 2024年3月～2024年5月：文献阅读。 * 2024年6月：开题，方案设计。 * 2024年7月：死代码探测技术方案设计，现有工具的改进工作。 * 2024年8月：进一步细化SNT的模块化粒度，明晰SNT的P2P网络实现。 * 2024年9月：将SNT的P2P网络模块逐步移植到AsyncOS内核。 * 2024年10月：将SNT基于P2P的一些简单功能，例如节点发现、已知节点管理等功能移植到rCAsyncOSore内核。 * 2024年11月：将SNT的更高层级的功能移植到AsyncOS，例如区块生产和敲定，账户信息管理等。 * 2024年12月～2025年2月：毕业论文撰写。 * 2025年3月～2025年6月：毕业论文完善和毕业答辩。 |

**五 预期研究成果**

|  |
| --- |
| 详细说明预期研究取得的成果，包括但不限于新理论、新方法、新技术以及新装置、新方案等,以及预期发表的论文、申请的专利、参加的学术会议等。   * 基于函数调用图的死代码探测技术方案一份。 * 移除死代码后的Substrate Node Template区块链软件一份。 * 至少包含区块链所需的p2p网络功能的AsyncOS操作系统一份。 * 描述以上两款软件的文档各一份。 |

**六 本课题创新之处**

|  |
| --- |
| 论证说明研究内容、拟采用的研究方法、技术路线或预期成果中有哪些创新之处。  本选题的创新点有：一是为rust编程语言实现了一种基于函数调用图的死代码探测技术；二是将经过死代码移除处理后的软件移植到另一操作系统中，以证明该技术对于移植工作的便利性贡献度。三是探索了已有rust软件产品与具体操作系统解耦并在新操作系统上重新兼容的可能解决方案，对嵌入式开发、软硬件平台完全自主可控开发等开发场景的软件生态扩展工作具有一定参考意义。 |

**七 研究基础**

|  |
| --- |
| 1.与本项目有关的研究工作积累和已取得的研究工作成绩。  已完成Rupta的改造，死代码检测已取得阶段性进展。  2.已具备的实验条件，尚缺少的实验条件和解决的途径（包括利用国家重点实验室和部门开放实验室的计划与落实情况）。  已具备实验需要的全部机具，无缺少的条件。  3. 研究经费预算计划和落实情况。  本研究无需经费。 |