Orizon Programming Language - 要件定義書

プロジェクトのビジョンと目標

核心ビジョン

Orizonは、現存するすべてのシステムプログラミング言語（C、C++、Rust、Go、Zigなど）を技術的に凌駕し、開発者体験、安全性、パフォーマンス、拡張性において世界一の地位を確立する革命的なシステムプログラミング言語です。

解決する根本的課題

1. **既存言語の根本的限界**
    - C/C++: メモリ安全性の欠如、未定義動作、複雑なビルドシステム
    - Rust: 学習曲線の急峻性、borrow checkerの制約、コンパイル時間
    - Go: システムレベルプログラミングでの制約、GCオーバーヘッド
    - Zig: エコシステムの未成熟、言語仕様の不安定性
2. **開発者体験の革新的改善**
    - 直感的で美しい構文設計
    - 世界最速のコンパイル速度
    - AI駆動の開発支援ツールチェーン
    - リアルタイム静的解析とサジェスト
3. **次世代システム要件への対応**
    - 量子コンピューティング時代への準備
    - 分散システムでのゼロオーバーヘッド並行性
    - エッジコンピューティング最適化
    - カーボンニュートラルな計算効率

長期的戦略目標

- **5年以内**: システムプログラミング分野でRustのシェアを上回る
- **10年以内**: OSカーネル、ブラウザエンジン、データベースエンジンの主要実装言語となる
- **15年以内**: TIOBE Indexでトップ3入り、全システムソフトウェアの標準言語化

対象ユーザー

主要ペルソナ

1. システムソフトウェアアーキテクト（上級）

- **背景**: 10年以上のC++/Rust経験、OSやブラウザエンジン開発
- **ニーズ**: 最高のパフォーマンスと安全性の両立、大規模プロジェクト管理
- **課題**: 既存言語での開発効率の限界、技術的負債の蓄積

2. 新世代システムプログラマー（中級）

- **背景**: 3-7年のモダン言語経験、クラウドネイティブ開発
- **ニーズ**: 学習しやすく強力なシステム言語、キャリア発展
- **課題**: Rustの学習困難性、C++の複雑性に対する懸念

3. 研究開発エンジニア（専門家）

- **背景**: アカデミック背景、高性能計算、AI/ML基盤開発
- **ニーズ**: 数値計算最適化、GPU/TPU統合、研究成果の実用化
- **課題**: 既存言語での数値計算表現力の限界

4. フルスタック開発者（拡張志向）

- **背景**: Web/モバイル開発からシステム領域への拡張希望
- **ニーズ**: 習得しやすいシステム言語、既存スキルの活用
- **課題**: システムプログラミングへの参入障壁の高さ

機能要件

コア言語機能

1. 革新的型システム

- **Dependent Types 2.0**: Rustの所有権システムを超える依存型システム
- **Linear Logic Integration**: リソース使用の数学的保証
- **Effect System**: 副作用の静的追跡と制御
- **Gradual Typing**: 動的型付けから静的型付けへの段階的移行サポート
- **Quantum Types**: 量子計算ステートの型レベル表現

2. メモリ管理の革命

- **Zero-Cost Garbage Collection**: コンパイル時完全解析による実行時オーバーヘッドゼロ
- **Region-Based Memory**: 自動的な領域推論とライフタイム管理
- **Memory Compression**: 実行時メモリ使用量の自動最適化
- **NUMA-Aware Allocation**: 分散メモリアーキテクチャでの最適化

3. 並行性とパラレリズム

- **Actor Model 3.0**: Erlang/Elixirを超える軽量プロセスシステム
- **Software Transactional Memory**: ハードウェア支援STMとの統合
- **Lock-Free Primitives**: 完全なロックフリーデータ構造ライブラリ
- **Async/Await Evolution**: ゼロオーバーヘッド非同期プログラミング

4. パフォーマンス最適化

- **Profile-Guided Optimization**: 実行時プロファイルによる自動最適化
- **Multi-Stage Programming**: コンパイル時計算とコード生成
- **Vectorization Auto-Detection**: SIMD命令の自動活用

- **Cache-Conscious Data Layout**: キャッシュ効率を考慮した自動データ配置

5. 安全性とセキュリティ

- **Information Flow Control**: 機密情報の流れの静的解析
- **Capability-Based Security**: 権限の細粒度制御
- **Cryptographic Types**: 暗号学的プリミティブの型レベル保証
- **Formal Verification Integration**: 定理証明器との直接統合

開発者体験機能

1. IDE統合とツールチェーン

- **Universal Language Server**: 全エディタ/IDE対応のAI駆動開発支援
- **Real-time Error Prevention**: 入力と同時の静的解析とサジェスト
- **Intelligent Refactoring**: AI駆動の大規模コードベース変更支援
- **Visual Debugging**: 並行処理やメモリ使用の可視化デバッグ

2. パッケージ管理とエコシステム

- **Reproducible Builds**: 完全な再現可能ビルドの保証
- **Zero-Config Package Management**: 設定レスなパッケージ管理
- **Cross-Platform Distribution**: ワンクリックマルチプラットフォーム配布
- **Semantic Versioning 2.0**: 破壊的変更の自動検出と段階的移行

3. テストとQA

- **Property-Based Testing**: QuickCheckスタイルテストの標準統合
- **Mutation Testing**: コードの変更に対するテストの堅牢性検証
- **Fuzz Testing Integration**: ファジングテストの標準ワークフロー統合
- **Formal Specification**: 仕様記述からのテスト自動生成

システム統合機能

1. 既存システムとの相互運用性

- **C ABI Compatibility**: 既存Cライブラリの直接呼び出し（C依存を避けつつ）
- **Foreign Function Interface**: 他言語との効率的な相互運用
- **Legacy Code Migration**: 既存コードベースの段階的移行支援
- **Standard Library Mapping**: 他言語標準ライブラリの自動マッピング

2. プラットフォーム支援

- **WebAssembly Native**: WASM-GCを活用したウェブネイティブ実行
- **GPU Computing**: CUDA/OpenCL/Vulkan Computeの統合抽象化
- **Embedded Systems**: マイクロコントローラから組み込みLinuxまで対応
- **Cloud Native**: コンテナ、サーバーレス環境での最適化

非機能要件

パフォーマンス要件

1. コンパイル性能

- **コンパイル速度**: Goの2倍、Rustの10倍の速度を達成
- **インクリメンタルコンパイル**: 変更行数に比例した線形コンパイル時間
- **並列コンパイル**: 全CPUコアを効率活用したビルド並列化
- **キャッシュ最適化**: インテリジェントなビルドキャッシュによる再コンパイル最小化

2. 実行時パフォーマンス

- **CPU効率**: C++並みの計算効率（±5%以内）
- **メモリ効率**: Rustと同等以上のメモリ安全性を持ちつつ、10%以上の省メモリ化
- **起動時間**: JITコンパイルなしでの瞬間起動（<1ms）
- **レイテンシ**: リアルタイムシステム要件を満たす予測可能な応答時間

3. スケーラビリティ

- **並行スケーリング**: 1024コアまでの線形スケーリング
- **分散処理**: ネットワーク越しでの透明な並列処理
- **メモリスケーリング**: テラバイト級メモリでの効率的な処理

セキュリティ要件

1. メモリ安全性

- **バッファオーバーフロー**: コンパイル時完全防止
- **Use-After-Free**: 型システムレベルでの根絶
- **Double-Free**: 線形型による自動防止
- **データ競合**: 並行アクセスの静的検証

2. 暗号学的セキュリティ

- **Timing Attack Resistance**: 定数時間実行の型レベル保証
- **Side-Channel Protection**: サイドチャネル攻撃への自動対策
- **Quantum-Safe Cryptography**: 量子計算攻撃耐性の標準サポート
- **Secure Multi-Party Computation**: SMCプリミティブの言語内統合

3. ソフトウェアサプライチェーン

- **Code Signing**: ソースコードから実行バイナリまでの署名チェーン
- **Reproducible Builds**: ビット単位で同一なビルド結果の保証
- **Dependency Verification**: 依存パッケージの完全性検証
- **Vulnerability Scanning**: 既知脆弱性の自動検出とアラート

信頼性・可用性要件

1. エラーハンドリング

- **Total Error Coverage**: すべてのエラー状態の静的検証
- **Graceful Degradation**: 部分的障害での段階的機能低下
- **Circuit Breaker Pattern**: 分散システムでの障害波及防止
- **Error Recovery**: 自動的なエラー回復メカニズム

2. 監視と観測可能性

- **Built-in Telemetry**: 実行時メトリクスの標準的な取得
- **Distributed Tracing**: マイクロサービス間の処理追跡
- **Performance Profiling**: 本番環境でのゼロオーバーヘッドプロファイリング
- **Health Monitoring**: システム健全性の継続的な監視

保守性・拡張性要件

1. コードの可読性と保守性

- **Self-Documenting Code**: コード自体が仕様書となる表現力
- **Automated Documentation**: コードからの自動ドキュメント生成
- **Code Analysis**: 技術的負債と保守性指標の自動計測
- **Refactoring Safety**: 型安全性を保った大規模リファクタリング

2. 言語の進化と互換性

- **Backward Compatibility**: 既存コードの長期サポート保証
- **Feature Flags**: 新機能の段階的導入メカニズム
- **Migration Tools**: 言語バージョン間の自動移行支援
- **Extension Points**: サードパーティによる言語拡張の標準化

ユーザビリティ要件

1. 学習しやすさ

- **Progressive Disclosure**: 段階的な機能習得をサポートする設計
- **Familiar Syntax**: 既存言語経験者にとって直感的な構文
- **Interactive Learning**: REPL環境での学習支援
- **Error Messages**: 初心者にも理解しやすいエラーメッセージ

2. 開発効率

- **Zero-Config Development**: 設定なしでの即座な開発開始
- **Hot Reloading**: コード変更の即座な反映
- **Intelligent Autocomplete**: AI駆動のコード補完
- **Template Generation**: プロジェクト種別に応じたテンプレート自動生成

環境要件

1. 実行環境

- **Operating Systems**: Windows, macOS, Linux (x86_64, ARM64)

- **WSL Compatibility**: Windows Subsystem for Linux完全対応
- **Container Support**: Docker, Podman, containerdネイティブ対応
- **Cloud Platforms**: AWS, GCP, Azure, Oracle Cloudでの最適化

2. 開発環境

- **Hardware Requirements**: 最小4GB RAM、推奨16GB RAM
- **Editor Support**: VS Code, IntelliJ, Vim/Neovim, Emacs
- **CI/CD Integration**: GitHub Actions, GitLab CI, Jenkins, CircleCI
- **Debugger Support**: GDB, LLDB互換デバッガーインターフェース

制約事項

技術的制約

1. C/C++依存の完全排除

- **コンパイラ実装**: C/C++で書かれたLLVM等への依存を避け、純粋な代替実装を採用
- **システムライブラリ**: libcやglibc等のC標準ライブラリに依存しない独自実装
- **ツールチェーン**: GCC, Clangに依存しない独立したコンパイラ実装
- **バイナリ互換性**: C ABIとの互換性を保ちつつ、実装レベルでのC依存を排除

2. 実装制約

- **Bootstrap Strategy**: セルフホスティング（自分自身でコンパイルできる）の早期実現
- **Memory Footprint**: コンパイラ自体のメモリ使用量最適化（<512MB）
- **Compilation Model**: AOT (Ahead-of-Time) コンパイルを主軸とした設計
- **Target Architecture**: x86_64, ARM64, RISC-V, WebAssemblyへの対応

3. エコシステム制約

- **Package Registry**: 中央集権的でないパッケージ配布システム
- **Version Management**: セマンティックバージョニングの厳格な適用
- **License Compatibility**: オープンソースライセンスとの完全な互換性
- **Standard Library**: 最小限のコアライブラリから段階的な拡張

プロジェクト制約

1. 開発リソース

- **開発期間**: 24-30ヶ月での最初のプロダクションリリース
- **チーム構成**: コンパイラ、言語設計、ツールチェーン、エコシステムの専門チーム
- **品質基準**: 企業環境での採用に耐える安定性と信頼性
- **ドキュメント**: 包括的なドキュメントとチュートリアルの整備

2. 採用戦略制約

- **Interoperability**: 既存システムとの段階的統合を可能にする設計
- **Migration Path**: 他言語からの移行コストを最小化する支援ツール

- **Community Building**: オープンで包括的な開発者コミュニティの構築
- **Enterprise Adoption**: エンタープライズ環境での採用に必要な機能とサポート

3. 持続可能性制約

- **Carbon Neutrality**: エネルギー効率的なコンパイル・実行の実現
- **Open Source Strategy**: 持続可能なオープンソース開発モデルの確立
- **Vendor Independence**: 特定のベンダーに依存しない技術スタック
- **Long-term Support**: 10年以上の長期サポート計画