# witnz 実装計画 ver1

PostgreSQL 分散改竄検知システム

## 1. プロジェクト構成

```
witnz/
├── cmd/witnz/main.go
├── internal/
│   ├── cdc/           # PostgreSQL CDC
│   ├── consensus/     # Raft
│   ├── hash/          # HashChain, MerkleTree
│   ├── storage/       # BoltDB
│   ├── verify/        # 検証ロジック
│   └── config/        # 設定
├── docker-compose.yml
├── Dockerfile
├── go.sum
├── go.mod
└── Makefile
```

### 依存ライブラリ

| ライブラリ | 用途 |
|---|---|
| hashicorp/raft | 分散コンセンサス |
| jackc/pglogrepl | PostgreSQL Logical Replication |
| jackc/pgx/v5 | PostgreSQL ドライバ |
| etcd-io/bbolt | ローカル KV ストア（Raft log + hash storage） |
| grpc/grpc-go | ノード間通信 |
| spf13/cobra + viper | CLI + 設定管理 |

# 2. CDC モジュール

PostgreSQL Logical Replication でテーブル変更を検知する。

### データ構造

```
type ChangeEvent struct {
    Table     string
    Operation OpType  // INSERT, UPDATE, DELETE
    LSN       uint64
    OldTuple  map[string]any
    NewTuple  map[string]any
    Timestamp time.Time
}
```

### インターフェース

```
type Listener interface {
    Start(ctx context.Context) error
    Stop() error
    Events() <-chan *ChangeEvent
}
```

### 実装方針

- `pglogrepl.StartReplication` で Replication Slot 接続
- `pgoutput` プラグイン使用（PostgreSQL 10+標準）
- Publication 自動作成: `CREATE PUBLICATION witnz FOR TABLE ...`
- Relation/Insert/Update/Delete メッセージをデコード → ChangeEvent 変換
- LSN 永続化でクラッシュリカバリ対応
- StandbyStatusUpdate 定期送信（タイムアウト防止）

# 3. Hash モジュール

## 3.1 Hash Chain（append_only モード）

監査ログなど追記のみのテーブル用。過去レコードの改竄を検知。

```
type ChainEntry struct {
    Index    uint64
    PrevHash [32]byte
    DataHash [32]byte  // SHA-256(row)
    Hash     [32]byte  // SHA-256(Index || PrevHash || DataHash)
    LSN      uint64
}
type HashChain struct {
    func Append(data []byte, lsn uint64) *ChainEntry
    func Get(index uint64) *ChainEntry
    func Head() *ChainEntry
    func Verify() (valid bool, brokenAt uint64)
}
```

## 3.2 Merkle Tree（state_integrity モード）

権限テーブルなど UPDATE 可能なテーブル用。現在値の改竄を検知。

```
type MerkleTree struct {
    func Root() [32]byte
```

```
    func Insert(pk string, data []byte)
    func Update(pk string, data []byte)
    func Delete(pk string)
    func Proof(pk string) []ProofNode
    func Verify(pk string, data []byte, proof []ProofNode) bool
}
```

## 4. Consensus モジュール（Raft）

hashicorp/raft を使用。ハッシュ状態を複数ノード間で複製・合意する。

### FSM（Finite State Machine）

```
type WitnzFSM struct {
    chains  map[string]*hash.HashChain   // table -> chain
    merkles map[string]*hash.MerkleTree  // table -> tree
    store   storage.Store
}

// raft.FSM interface
func (f *WitnzFSM) Apply(log *raft.Log) interface{}
func (f *WitnzFSM) Snapshot() (raft.FSMSnapshot, error)
func (f *WitnzFSM) Restore(rc io.ReadCloser) error
```

### LogCommand

```
type CommandType uint8
const (
    CmdAppendHash CommandType = iota  // Hash Chain 追加
    CmdUpdateMerkle                   // Merkle Tree 更新
)

type LogCommand struct {
    Type       CommandType
    Table      string
    ChainEntry *hash.ChainEntry  // for CmdAppendHash
    MerkleOp   *MerkleOp         // for CmdUpdateMerkle
}
```

### Raft 設定

```
config := raft.DefaultConfig()
config.LocalID = raft.ServerID(nodeID)
config.HeartbeatTimeout = 1 * time.Second
config.ElectionTimeout  = 1 * time.Second
config.SnapshotInterval = 120 * time.Second

// Transport: TCP or gRPC
// LogStore, StableStore: raftboltdb.NewBoltStore()
// SnapshotStore: raft.NewFileSnapshotStore()
```

## 5. Storage モジュール

BoltDB を使用。Raft log + hash state を永続化。

```
type Store interface {
    // Hash Chain
    AppendChainEntry(table string, entry *ChainEntry) error
```

```
    GetChainEntry(table string, index uint64) (*ChainEntry, error)
    GetChainHead(table string) (*ChainEntry, error)

    // Merkle Tree
    SaveMerkleSnapshot(table string, tree *MerkleTree) error
    LoadMerkleSnapshot(table string) (*MerkleTree, error)

    // LSN tracking
    SaveLSN(lsn uint64) error
    GetLSN() (uint64, error)
}
```

### BoltDB Bucket 構成

```
chains/          # Hash Chain entries
  {table}/
    {index} -> ChainEntry (gob/protobuf)
merkle/          # Merkle Tree snapshots
  {table} -> serialized tree
meta/            # LSN, config
  lsn -> uint64
```

## 6. 検証ロジック

### 6.1 append_only 検証

```
func VerifyAppendOnly(table string) (*VerifyResult, error) {
    // 1. DB からレコード全件取得（ORDER BY pk）
    // 2. 各行のハッシュを計算
    // 3. ローカル HashChain と比較
    // 4. 不一致があれば他ノードに問い合わせ
    // 5. 多数決で正しい状態を決定
}
```

### 6.2 state_integrity 検証

```
func VerifyStateIntegrity(table string) (*VerifyResult, error) {
    // 1. DB からレコード全件取得
    // 2. Merkle Tree 再構築 → Root 計算
    // 3. ローカルの Merkle Root と比較
    // 4. 不一致時、Proof で改竄レコード特定
    // 5. 他ノードの Root と照合
}
```

### 6.3 多数決アルゴリズム

```
func ConsensusCheck(table string, localHash [32]byte) bool {
    votes := map[[32]byte]int{localHash: 1}
    for _, peer := range cluster.Peers() {
        h := peer.GetHash(table)
        votes[h]++
    }
    majority := (len(cluster.Peers())+1)/2 + 1
    return votes[localHash] >= majority
}
```

# 7. データフロー

## 書き込み時

```
App → PostgreSQL INSERT
    ↓ Logical Replication
CDC Listener → ChangeEvent
    ↓
Hash 計算 → LogCommand 作成
    ↓ Raft Apply (Leader only)
FSM.Apply() → HashChain/MerkleTree 更新
    ↓ Raft Replication
全ノードで状態同期
```

## 検証時

```
定期 or CLI verify
    ↓
PostgreSQL から対象テーブル読み込み
    ↓
ハッシュ再計算 → ローカル状態と比較
    ↓ 不一致時
他ノードに問い合わせ → 多数決判定
    ↓ 改竄検知
アラート発報（Slack/Webhook）
```

# 8. 設定ファイル

```yaml
# witnz.yaml
node:
  id: node-1
  data_dir: /var/lib/witnz
  bind: 0.0.0.0:7300

cluster:
  peers:
    - id: node-1
      addr: 10.0.0.1:7300
    - id: node-2
      addr: 10.0.0.2:7300

database:
  dsn: postgres://user:pass@host:5432/db
  slot: witnz_slot

tables:
  - name: audit_logs
    mode: append_only
  - name: permissions
    mode: state_integrity
    verify_interval: 60s
```

## 9. CLI コマンド

```
witnz init      # 初期化、Publication/Slot 作成
witnz start     # ノード起動
witnz status    # クラスタ状態表示
witnz verify    # 即時検証実行
witnz protect <table> --mode=append_only
```

*— End —*