github：https://github.com/skip-mev/block-sdk
文档：https://docs.skip.money/blocksdk/overview/
private mempool ：https://github.com/skip-mev/mev-tendermint
Lane app store：https://docs.skip.money/blocksdk/lanes/existing-lanes/mev：mev、free、Custom Fee Market
Skipper Bot-船长机器人：https://github.com/skip-mev/skipper

MEV Lane ：允许在协议内进行区块顶部 MEV 拍卖，并将收入重新分配给区块链。
  集成步骤：https://docs.skip.money/blocksdk/lanes/existing-lanes/mev
Free Lane：自由通道允许某些交易包含在一个区块中而无需支付费用。 该通道可用于鼓励链上的某些行为，例如质押、治理或其他行为。
  集成步骤：https://docs.skip.money/blocksdk/lanes/existing-lanes/free
Default Lane：默认通道按交易费用对交易进行排序。
  集成步骤：https://github.com/skip-mev/block-sdk/tree/main/lanes/base

Block SDK 是一组 Cosmos SDK 和 ABCI++ 原语，允许链根据特定用例完全自定义块。
它将您的链的块变成具有自己特殊功能的**`highway`**个体的组合。**`lanes`**

`lanes`Skip在 SDK 上构建了许多您的协议可以使用的即插即用功能，包括协议内 MEV 重新捕获和预言机！
此外，可以扩展 Block SDK 以添加**您自己的自定义`lanes`**来配置您的块，以完全满足您的应用程序需求。

您可以将 Block SDK 视为一个**交易`highway`系统**，其中 `lane`高速公路上的每个交易系统都有特定的用途，并且有自己的一套规则和交通流量。

在Block SDK中，各`lane`有一套自己的规则和交易流程管理系统。

- A`lane`是我们传统上认为的标准内存池，其中共享交易的交易***验证***、***排序***和***优先级。***
- `lanes`实现一个**标准接口**，允许每个人`lane`提出并验证一个区块的一部分。
- `lanes`相互订购，由开发人员配置。所有这些`lanes` 一起定义了所需的链块结构。

Block SDK 用例

带有分离的块`lanes`可用于：

1. **MEV 缓解**：可以设计区块顶部通道来创建协议内区块顶部拍卖，以透明且可治理的方式重新捕获 MEV。
2. **免费/减费交易**：具有某些属性的交易（例如来自受信任的帐户或执行鼓励的操作）可以利用免费通道来促进*良好的*行为。
3. **专用预言机空间**可以在其他类型的交易之前包含预言机，以确保价格更新首先发生，并且不能被夹在中间或被操纵。
4. **订单流拍卖**：可以构建一个 OFA 通道，以便订单流提供商可以将其提交的交易与特定的后备者捆绑在一起，以保证 MEV 奖励归还给用户
5. **增强和可定制的隐私**：可以引入隐私增强功能，例如阈值加密通道，以保护用户数据并维护特定用例的隐私。
6. **费用市场改进**：一个或多个费用市场（例如 EIP-1559）可轻松适用于不同的通道（可能为某些 dApp 定制）。每个智能合约/交易所都可以拥有自己的费用市场或拍卖，用于交易排序。
7. **拥塞管理**：将交易分段到通道可以通过限制某些应用程序的使用和定制费用市场来帮助缓解网络拥塞。

Application Mempools-应用内存池：
在 cosmos-sdk v0.47.0 中，应用程序端内存池被添加到 SDK 中。
借助应用程序端内存池，验证者不再需要依赖共识引擎来跟踪和排序所有可用交易。
现在应用程序可以定义自己的内存池实现，即
  1、存储所有待处理（未在块中最终确定）的交易
  2、对待处理交易集进行排序

区块构建如何变化?
现在，在PrepareProposal中，验证者可以从其应用程序状态感知内存池中提取交易，
而不是从共识引擎获取交易，并优先考虑这些交易而不是共识引擎的交易。

为什么这样更好?
  1、不支持应用程序状态的内存池将无法制定支持状态的排序规则。
    A、所有抵押者交易都放置在区块的顶部
    B、所有 IBC LightClientUpdate 消息都放置在块的顶部
    C、你能想到的都可以！！
  2、共识引擎的内存池通常效率低下。
    A、共识引擎的内存池不知道何时从自己的内存池中删除交易
    B、共识引擎花费大部分时间在对等点之间重新广播交易，占用网络带宽

Block-SDK!!
Block-SDK 定义了自己的应用程序端内存池的自定义实现，即 LaneMempool。
LaneMempool 由 Lane 组成，处理事务入口、排序和清理。

交易入口
  1、LaneMempool 构造函数定义了通道的顺序。 当应用程序收到交易时，它会按顺序迭代所有通道并将交易插入到其所属的第一个通道中。
  2、LanedMempool 的每个 Lane 都维护自己的交易顺序。
    当 LanedMempool 将交易路由到其相应的 Lane 时，Lane 会将该交易插入到相对于该 Lane 中所有其他交易的指定位置

PrepareProposal-准备提案
当应用程序被指示进行PrepareProposal时，它会按顺序迭代其通道，并调用每个通道的PrepareLane方法。
Lane.PrepareLane 方法从 Lane 收集事务，并将这些事务附加到之前 Lane 的PrepareLane 调用的事务集中。
换句话说，每个区块提案现在都是来自 LanedMempool 组成通道的交易的集合。

ProcessProposal-流程提案
当应用程序收到提案并调用 ProcessProposal 时，应用程序会将验证委托给 LaneMempool.ProcessLanes 方法。
请记住，提案由来自 LaneMempool 子通道的交易组成，因此，LaneMempool 可以将每个 Lane 对提案的贡献路由到该 Lane 进行验证。
当莱恩的所有贡献均有效时，该提案就会通过。

备注：由LaneMempool的PrepareLanes方法构造的块必须始终通过该LaneMempool的ProcessLanes方法，否则，该链将无法产生块！
这些功能对于达成共识至关重要，因此在实现它们时要小心！！

Default Lane：

为了进行设置，我们将实施默认通道，这是接受所有交易的最通用且限制最少的通道。
这不会导致您的链功能发生任何变化，但会让您为之后添加具有更多功能的通道做好准备！

默认通道反映了 CometBFT 目前创建提案的方式。
  1、它进行基本检查以确保交易有效。
  2、根据交易费用金额（从最高到最低）对交易进行排序。
  3、PrepareLane 处理程序将从通道中获取交易，直至达到 MaxBlockSpace 限制
  4、ProcessLane 处理程序将确保交易根据其费用金额进行排序，并通过在PrepareLane 中完成的相同检查。

集成步骤：
1、 Create the lanes

```
import (
"github.com/skip-mev/block-sdk/abci"
"github.com/skip-mev/block-sdk/block/base"
defaultlane "github.com/skip-mev/block-sdk/lanes/base"
)

// 1. Create the lanes.
// NOTE: The lanes are ordered by priority. The first lane is the highest priority
// lane and the last lane is the lowest priority lane. Top of block lane allows
// transactions to bid for inclusion at the top of the next block.

// For more information on how to utilize the LaneConfig please
// visit the README in docs.skip.money/chains/lanes/build-your-own-lane#-lane-config.

// Default lane accepts all transactions.
func NewApp() {
...
defaultConfig := base.LaneConfig{
    Logger: app.Logger(),
    TxEncoder: app.txConfig.TxEncoder(),
    TxDecoder: app.txConfig.TxDecoder(),
    MaxBlockSpace: math.LegacyZeroDec(),
    MaxTxs: 0,
}
defaultLane := defaultlane.NewDefaultLane(defaultConfig)
// TODO(you): Add more Lanes!!!
```

2、SetMempool：在您的基本应用程序中，您将需要创建一个`LanedMempool`由`lanes`您想要使用的组成的。

```
// 2. Set up the relative priority of lanes
lanes := []block.Lane{defaultLane,}

mempool := block.NewLanedMempool(app.Logger(), true, lanes...)

app.App.SetMempool(mempool)
```

3、ante handler：接下来，按优先级对车道进行排序。第一个车道是最高优先级车道，最后一个车道是最低优先级车道。**建议将最后一个车道作为默认车道。**

```
// 3. Set up the ante handler.
anteDecorators := []sdk.AnteDecorator{
    ante.NewSetUpContextDecorator(),
    ...

    utils.NewIgnoreDecorator(
        ante.NewDeductFeeDecorator(
            options.BaseOptions.AccountKeeper,options.BaseOptions.BankKeeper,
            options.BaseOptions.FeegrantKeeper,options.BaseOptions.TxFeeChecker,
        ),
        options.FreeLane,
    ),
...
}

anteHandler := sdk.ChainAnteDecorators(anteDecorators...)

// Set the lane ante handlers on the lanes.
// NOTE: This step is very important. Without the antehandlers, lanes will not be able to verify transactions.
for _, lane := range lanes {
    lane.SetAnteHandler(anteHandler)
}

app.App.SetAnteHandler(anteHandler)
```

4、LanedMempool's ProposalHandler：您还需要创建一个PrepareProposalHandler 和一个ProcessProposalHandler 分别负责准备和处理提案。
  配置PrepareProposalHandler和ProcessProposalHandler中通道的顺序，以匹配LanedMempool中通道的顺序。

```
// 4. Set the abci handlers on base app
// Create the LanedMempool's ProposalHandler
proposalHandler := abci.NewProposalHandler(app.Logger(),app.TxConfig().TxDecoder(),mempool,)

// set the Prepare / ProcessProposal Handlers on the app to be the `LanedMempool`'s
app.App.SetPrepareProposal(proposalHandler.PrepareProposalHandler())
app.App.SetProcessProposal(proposalHandler.ProcessProposalHandler())
```

怎么运行的：
概括
使用 Block SDK，块被分解为更小的部分块，称为通道。
  1、每个通道都有自己的自定义区块构建逻辑并存储不同类型的交易。
  2、每个通道只能消耗通道配置 (MaxBlockSpace) 中定义的块的一部分。
  3、当请求区块提案时，区块将按照应用程序中定义通道的顺序迭代地填充来自每个通道的交易。
  4、处理块提案时，每个通道将按照应用程序中定义通道的顺序迭代验证其块部分。
  5、区块中的交易必须尊重通道的顺序。

交易生命周期：了解一般交易生命周期对于理解通道如何工作非常重要。
  1、当交易被签名并广播到链上的节点时，交易就开始了。
  2、然后它将由节点上的应用程序进行验证。
  3、如果有效，它将被插入到节点的内存池中，内存池是包含在块中之前用于交易的存储区域。
  4、如果该节点恰好是验证者，并且正在提议一个块，则应用程序将调用PrepareProposal来创建一个新的块提议。
  5、提议者将查看他们的内存池中有哪些交易，迭代选择交易直到块已满，并与其他验证者共享提议。
  6、当不同的验证者收到提案时，验证者将在签名之前通过 ProcessProposal 验证其内容。
  7、如果提案有效，验证者将签署该提案并将其投票广播到网络。
  8、如果该区块无效，验证者将拒绝该提案。
  9、一旦提案被网络接受，它就会作为一个区块提交，并且其中包含的交易将从每个验证者的内存池中删除。

Lane生命周期: 通道在上述交易生命周期中引入了新步骤。
  LaneMempool 由几个不同的通道组成，这些通道存储自己的交易。 LanedMempool 会将交易插入到所有接受它的通道中
  1、基础应用程序接受交易后，将检查该交易是否可以进入任何通道，如通道的 MatchHandler 所定义。
  2、Lane 可以配置为仅接受符合特定条件的交易。 例如，通道可以配置为仅接受与质押相关的交易（例如免费交易通道）。
  3、当提出新块时，应用程序的PrepareProposalHandler将迭代地调用每个通道上的PrepareLane。PrepareLane 方法与PrepareProposal 类似。
  4、在通道上调用PrepareLane将触发通道从其内存池中获取交易并将它们添加到提案中。
  5、当其他验证器在 ProcessProposal 中验证提案时，
      abci/abci.go 中定义的 ProcessProposalHandler 将以与在 PrepareProposalHandler 中调用的顺序相同的顺序调用每个通道上的 ProcessLane。
  6、对 ProcessLane 的每次后续调用都会过滤掉属于先前通道的事务。 给定通道的 ProcessLane 将仅验证属于该通道的事务。

当需要提出新块时，PrepareProposalHandler 将首先调用 LaneA 上的PrepareLane，然后调用 LaneB。

当 LaneA 上调用PrepareLane 时，LaneA 将从其内存池中获取交易并将其添加到提案中。
LaneB 也同样如此。 假设 LaneA 获取事务 Tx1 和 Tx2，LaneB 获取事务 Tx3 和 Tx4。
这给了我们一个由以下内容组成的提案：Tx1, Tx2, Tx3, Tx4

当ProcessProposalHandler被调用时，它会调用LaneA上的ProcessLane，提案由Tx1、Tx2、Tx3和Tx4组成。
LaneA 随后将验证 Tx1 和 Tx2 并返回剩余交易 - Tx3 和 Tx4。
然后，ProcessProposalHandler 将使用剩余事务（Tx3 和 Tx4）调用 LaneB 上的 ProcessLane。
LaneB 随后将验证 Tx3 和 Tx4，并且不返回任何剩余交易。

使用案例：交易拍卖出价

Default Auction Bid Message

```
// MsgAuctionBid defines a request type for sending bids to the x/auction module.

type MsgAuctionBid struct {
// bidder is the address of the account that is submitting a bid to the auction.
Bidder string

// bid is the amount of coins that the bidder is bidding to participate in the auction.
Bid types.Coin

// transactions are the bytes of the transactions that the bidder wants to bundle together.
Transactions [][]byte
}
```

Nonce Checking-随机数检查
一般来说，所有捆绑包都必须遵守帐户的随机数排序。 如果提交的包带有无效的随机数，它将被拒绝。
捆绑包的执行将始终如下所示：
  1、拍卖交易（提取出价）
  2、捆绑中的所有交易（按顺序）

例如，假设以下情况：
  1、搜索者拥有账户 A，且随机数为 n
  2、搜索者想要从帐户 A 提交包含 3 笔交易的捆绑包
  3、搜索者必须首先使用随机数 n + 1 签署 AuctionTx。
      搜索者必须使用随机数 n + 2 签署捆绑包中的第一笔交易，
      使用随机数 n + 3 签署第二笔交易，
      使用随机数 n + 4 签署第三笔交易。

Skipper Bot-船长机器人：https://github.com/skip-mev/skipper
用户可以利用 Skip 自己的开源 Skipper Bot 来引导他们的搜索机器人。

```
// createBidTx creates an auction bid tx with the given parameters

func createBidTx(privateKey *secp256k1.PrivKey,bidder,bid sdk.Coin,bundle [][]byte,height uint64,sequenceOffset uint64,){

    // bid transaction can only include a single MsgAuctionBid
    msgs := []sdk.Msg{&buildertypes.MsgAuctionBid{Bidder:bidder,Bid:bid,Transactions: bundle,},}


    // Retrieve the expected account and sequence number
    sequenceNum, accountNum := getAccountInfo(privateKey)

    txConfig := authtx.NewTxConfig(codec.NewProtoCodec(codectypes.NewInterfaceRegistry()),authTx.DefaultSignModes,)

    txBuilder := txConfig.NewTxBuilder()

    // Set the messages along with other fee information
    txBuilder.SetMsgs(msgs...)
    ...

    txBuilder.SetGasLimit(5000000)

    // SET A TIMEOUT HEIGHT FOR HOW LONG THE BID IS VALID FOR
    txBuilder.SetTimeoutHeight(height)


    // Sign the transaction with nonce n + 1 where n is the number of transactions in the bundle that are signed by the searcher
    offsetNonce := sequenceNum + sequenceOffset

    signerData := auth.SignerData{ChainID:CHAIN_ID,AccountNumber: accountNumber,Sequence: offsetNonce,}


    sigV2 = clientTx.SignWithPrivKey(signerData,txBuilder,privateKey,txConfig,offsetNonce,)

    txBuilder.SetSignatures(sigV2);
    return txBuilder.GetTx(),
}
```

拍卖费用：
所有拍卖参数均可通过标准节点上的 /block-sdk/x/auction/v1/params HTTP 路径或 x/auction 定义的 gRPC 服务进行访问。

为了参加拍卖，搜索者必须支付费用。 该费用以链上的原生代币支付。 费用由链设置的拍卖参数决定。 拍卖参数为：
  1、MaxBundleSize：指定捆绑包中可以包含的最大交易数量（捆绑包 = 交易的有序列表）。 捆绑包必须≤此数字。
  2、ReserveFee：指定参与拍卖的底价。 低于保留费的出价将被忽略。
  3、MinBidIncrement：指定每个后续出价必须大多少（如单个节点所见）才能被考虑。 如果出价低于当前最高出价+最小出价增量，则忽略该出价。
  4、FrontRunningProtection：决定是否启用抢先交易和三明治保护。

领先和夹层保护
如果设置为 true，您的捆绑包必须遵循以下准则：
  1、您必须将你已签名的交易放在你未签名(其他人签名)的交易之后
  2、捆绑包中最多只能有两个唯一签名者

捆绑包示例：

1. **有效**：[tx1, tx2, tx3]，其中 tx1 由签名者 1 签名，tx2 和 tx3 由投标人签名。
2. **有效**：[tx1, tx2, tx3, tx4]，其中 tx1 - tx4 由投标人签名。
3. **无效**：[tx1, tx2, tx3]，其中 tx1 和 tx3 由投标人签名，tx2 由其他签名者签名。 （可能是三明治攻击）
4. **无效**：[tx1, tx2, tx3]，其中 tx1 由投标人签名，tx2、tx3 由其他签名者签名。 （可能是抢先攻击）