Bitcoin Documentation

สมาชิกกลุ่ม

1. ธนกร	ชาญเชิงพานิช	62010336
2. ธนดล	สินอนันต์วณิช	62010345
3. ภูมิพงศ์	บุญดี	62010718
4. สิรวิชญ์	สุขวัฒนาวิทย์	62010948
5. สุทธิราช	ภูโท	62010966
6. สุรวิช	ยอแสง	62010966

Github source code : https://github.com/SAD-GROUPWORK/bitcoin

Bitcoin คืออะไร?

Bitcoin เป็นสกุลเงิน digital ประเภท cryptocurrency ที่ใช้เทคโนโลยี Blockchain มาใช้ในการทำธุรกรรมการเงิน โดยเป็นใครก็ได้จากทั่วทุกมุมโลก โดย Bitcoin core คือชื่อ open source software ที่ทำให้ bitcoin สามารถใช้งานได้

โดย Bitcoin ทำงานอยู่บน blockchain ซึ่งสร้างมาโดย concept decentralize เนื่องจากผู้คนเริ่มไม่ไว้ใจระบบ centralize อย่างเช่น ธนาคารในปัจจุบัน โดยธนาคารจะเป็นผู้รับความเสี่ยงทั้งหมดเอง แต่ทุกอย่างก็จะอยู่ในการควบคุมของธนาคารทั้งหมด ทำให้ถ้าวันนึงเกิดวิกฤตการเงินขึ้นมา หรือมีการแทรกแซงจากภาครัฐ ทรัพย์สินของเราอาจจะไม่ปลอดภัยอย่างยิ่ง ทำให้คนหันมาสนในระบบ blockchain ที่เราทุกคนเป็นผู้ดูแลทรัพย์สินของเราได้ ไม่มีการแทรกแซงจากผู้อื่นได้ มีข้อกำหนดทุกอย่างที่ทุกคนตกลงร่วมกัน (Consensus) และให้การยอมรับ

Architecture ของ Bitcoin

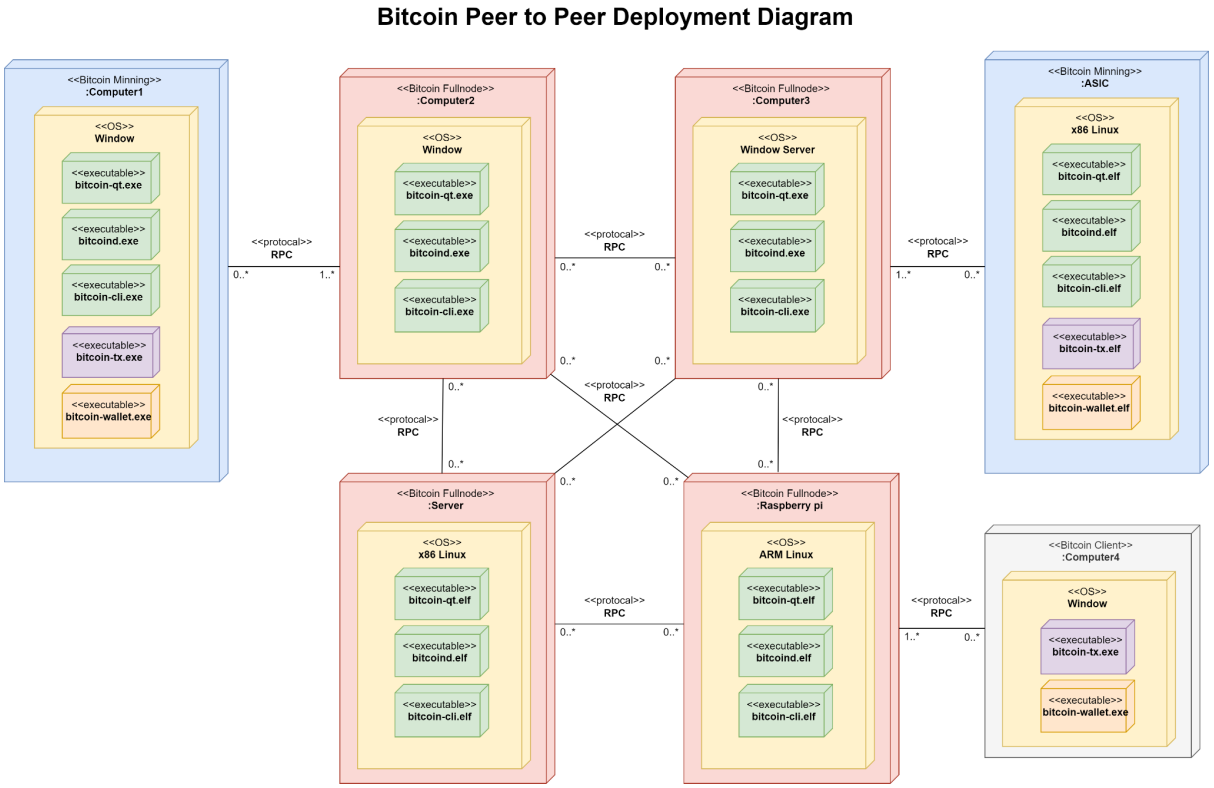
ตัวอย่าง Software Architecture Styles ที่เห็นชัดได้มากที่สุดของ Bitcoin คือ

Peer to Peer

Bitcoin จะใช้เทคโนโลยี Blockchain ในการยืนยันธุรกรรม ซึ่งเมื่อมีคนทำธุรกรรม เช่น การโอนเงินสกุล Bitcoin จะส่งข้อมูลของธุรกรรมนั้นไปยัง Bitcoin Node หลังจากนั้น Bitcoin Node จะนำธุรกรรมนั้น ไปเก็บไว้ที่ MemPool ซึ่งเป็นแหล่งเก็บธุรกรรมที่ยังไม่ได้รับการยืนยัน

หลังจากนั้น Miner จะดึงธุรกรรมที่อยู่ใน MemPool ไปยืนยันความถูกต้องของธุรกรรม เมื่อ Miner ยืนยันข้อมูลธุรกรรมนั้นเสร็จแล้ว จะส่งข้อมูลนั้นให้กับ Bitcoin Node เพื่อนำข้อมูลไปต่อลงใน Blockchain หลังจากนั้นจึงทำการกระจายข้อมูลบล็อกใหม่ไปยัง Node ข้างเคียงต่อไป แต่ละ Node ก็จะทำการเช็คข้อมูลกันเอง โดยการถือเสียงส่วนมากเป็นหลัก เพื่อพิสูจน์ความน่าเชื่อถือของข้อมูลในแต่ละ Node ว่ามีความถูกต้องจริงหรือไม่ จนเมื่อเวลาผ่านไป แต่ละ Node ที่เชื่อมด้วยกันแบบโครงข่าย Peer to Peer ซึ่งทุกๆ Node สามารถติดต่อสื่อสารกันได้ ก็จะมีข้อมูลที่ถูกต้องตรงกันในที่สุด

UML diagram ของ Architecture



Quality Attributes ของ Bitcoin

1. Security

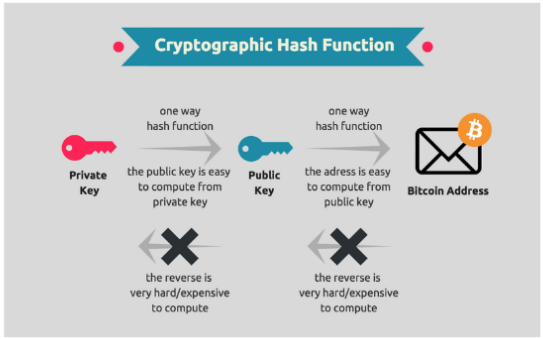
เนื่องจากข้อมูลบน Bitcoin network ไม่สามารถที่จะแก้ไขได้ หรือถ้าจะแก้ไขก็จำเป็นต้องใช้ทรัพยากรที่สูงมาก จนอาจจะไม่คุ้มค่าที่จะแก้ไขเลยก็ได้ ด้วยเทคโนโลยีในปัจจุบัน รวมทั้ง bitcoin ยังมี network ที่ใหญ่มากๆ จึงทำให้มีผู้ใช้งานอยู่ทั่วโลก ทำให้ยากต่อการที่จะมีกลุ่มคนใดกลุ่มคนนึง ช่วยกันแก้ไขข้อมูลบน Bitcoin network ได้ จึงทำให้ Bitcoin มีความปลอดภัยที่สูงมาก

Tactic for achieve Security

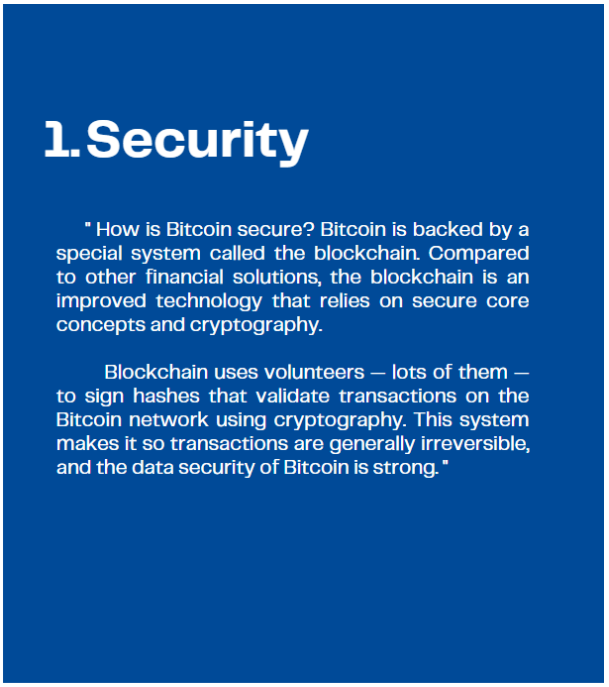
การเพิ่ม Security ของ Bitcoin จะทำได้โดยการมอบ Permission ที่ Node สามารถทำได้ ทำให้ Node มีข้อจำกัดในการเข้าถึงข้อมูลและไม่สามารถเข้าถึงข้อมูลอย่างอิสระได้

มีผู้ใช้งาน Blockchain จำนวนมากที่ได้ยืนยัน โดยการ sign hashes เพื่อยืนยันความถูกต้องของธุรกรรมการเงินที่เกิดขึ้นบน Bitcoin Network โดยใช้ ***Cryptography*** ทำให้การทำธุรกรรมการเงินต่างๆที่เกิดขึ้นบน Bitcoin Network จะไม่สามารถย้อนกลับได้ และทำให้เกิดความปลอดภัยของข้อมูลของ Bitcoin สูงมาก

Proof of Evidence



https://blockchainhub.net/blog/blog/cryptography-blockchain-bitcoin/
https://www.avg.com/en/signal/is-bitcoin-safe



Reference: https://www.avg.com/en/signal/is-bitcoin-safe

https://blockchainhub.net/blog/blog/cryptography-blockchain-bitcoin/

https://bitcoin.org/en/faq#security

2. Availability

Bitcoin ทำงานอยู่บนระบบ decentralized blockchain ซึ่งเป็นการกระจายการทำงานให้ผู้ใช้งาน Bitcoin ต่างจากระบบธนาคารในปัจจุบันที่ธนาคารเป็นผู้ดูแลทุกอย่าง ซึ่งเป็นแนวคิดแบบ centralized โดยข้อเสียของแนวคิดนี้คือ ถ้าวันนึงระบบมีปัญหา ทุกคนก็จะไม่สามารถใช้งานอะไรได้เลย ต่างจาก Bitcoin ที่มี Availability ที่สูงมาก

เนื่องจากมี node ของ Bitcoin ทำงานอยู่ทั่วโลก ทำให้ข้อมูลบน network สามารถเข้าถึงได้ตลอดเวลาจากทั่วทุกที่ โดยแค่ต้องติดต่อกับ node ใด node หนึ่งของ Bitcoin ก็ได้

ข้อมูลทั้งหมดเกี่ยวกับ Bitcoin จะอยู่ในระบบ Block chain เพื่อให้ทุกคนสามารถตรวจสอบและใช้งานได้แบบ Real-time ไม่มีบุคคลหรือองค์กรใดเข้ามาควบคุมหรือจัดการโปรโตคอลของ Bitcoin ได้ เนื่องจากมีความปลอดภัยในการเข้ารหัส สิ่งนี้ทำให้ระบบของ Bitcoin เชื่อถือได้ มีความโปร่งใส และสามารถคาดเดาได้อย่างสมบูรณ์

Tactic for achieve Availability

เงื่อนไข Availability ของ Bitcoin คือการที่คนให้การยอมรับ และมีการใช้งานที่สูง ทำให้มีคนสร้าง Bitcoin Node อยู่ทั่วโลก ซึ่งแต่ละ Node ติดต่อกันด้วยเครือข่ายแบบ Peer to Peer ดังนั้นการที่จะเข้ามาแทรกแซงหรือโจมตี Bitcoin นั้นทำได้ยากมาก ดังนั้นจุดแข็งของ Bitcoin คือการใช้งานที่สูง ในทางตรงกันข้าม เมื่อไม่มีใครใช้งาน Bitcoin ระบบก็จะสามารถโดนโจมตีหรือเข้าควบคุมได้ง่าย

Proof of Evidence

Reference: https://bitcoin.org/en/faq#what-are-the-advantages-of-bitcoin

3. Portability

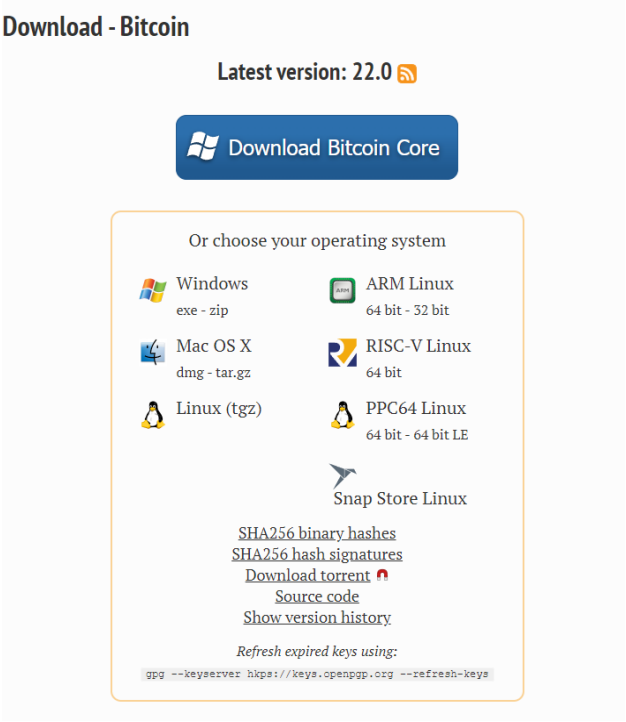
สำหรับตัว Bitcoin Core ได้มีการทำการ Compile ตัวโปรเจค Bitcoin ไว้รองรับทุกสถาปัตยกรรมของ Cpu ทั้ง Arm และ x86 อีกทั้งยังรองรับการทำงานแบบ 32-bit และ 64-bit

ดังนั้น Bitcoin Core จึงสามารถนำไปทำงานทั้งระบบ Windows, OS X และ ระบบปฏิบัติการ Linux หลายๆตัวที่ได้รับความนิยมได้ นอกจากนี้ผู้ใช้ยังสามารถนำ Bitcoin Core ไป compile สำหรับระบบปฏิบัติการ Linux ที่ตัวเองต้องการจะติดตั้งก็ได้เช่นกัน

Tactic for achieve Portability

ในโปรเจค Bitcoin นี้ ได้มีการพัฒนาโดยใช้ภาษา C/C++ ซึ่งเป็นภาษาที่รองรับสำหรับ Compilers ที่สามารถรันได้บน Platforms ต่างๆมากมายที่มีการรองรับตัวภาษานี้อยู่ โดยใช้ standard library C/C++ เพื่อที่จะทำให้สามารถรันบน Platforms ใดก็ได้ โดยไม่ต้องแก้ไขโค้ดเลยหรือมีการแก้ไขเพียงเล็กน้อยเท่านั้น

Proof of Evidence



Reference:

https://bitcoincore.org/en/download/

https://en.bitcoin.it/wiki/Bitcoin_Core_compatible_devices

https://bitcoin.org/en/full-node#minimum-requirements

https://www.cplusplus.com/info/description/#:~:text=...is%20portable..different%20platforms%20that%20support%20it.&text=C%2B%2B%20can%20use%20C%20libraries,modifications%20of%20the%20libraries'%20code.

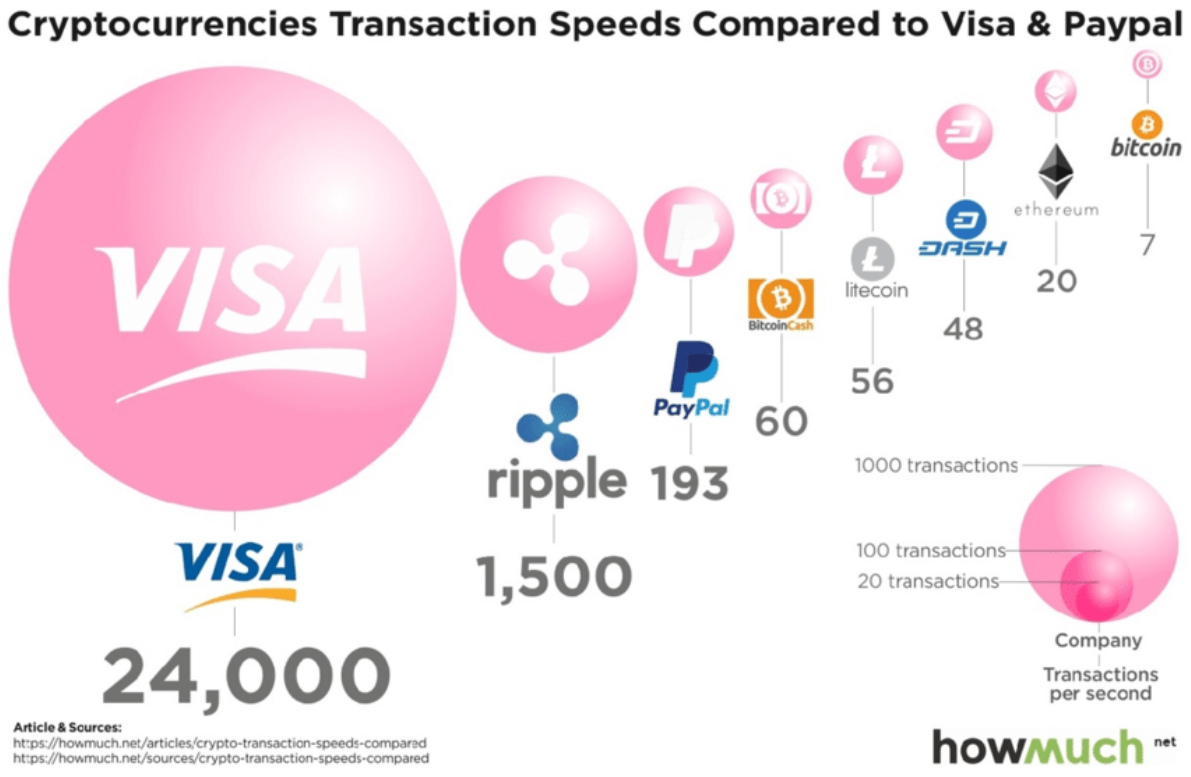
จุดอ่อนของ Architecture

Performance

จุดอ่อนของ Architecture ที่ใช้อยู่คือข้อจำกัดในเรื่องปริมาณการใช้งานของผู้ใช้ ซึ่ง Bitcoin รองรับ ON-CHAIN Transaction ได้ในปริมาณน้อย โดย Bitcoin สามารถทำธุรกรรมได้ 7 ธุรกรรมต่อวินาที โดยถ้าเทียบกับ VISA ซึ่งทำได้สูงโดยเฉลี่ยถึง 24000 ธุรกรรมต่อวินาที ทำให้เมื่อ Bitcoin มีการใช้งานที่สูงขึ้น มีธุรกรรมที่ต้องการยืนยันมากขึ้น Blockchain ก็จะเลือกแค่ธุรกรรมที่ให้ค่าธรรมเนียมสูงๆเท่านั้นมายืนยัน ส่งผลให้ธุรกรรมที่จ่ายค่าธรรมเนียมน้อย ก็จะล้มเหลวไปหรือทำธุรกรรมไม่สำเร็จ ด้วยสาเหตุนี้เมื่อ Bitcoin มีการใช้งานสูงขึ้น ค่าธรรมเนียมธุรกรรมก็จะสูงขึ้นตามไปด้วย

Tactic for achieve Performance

ในการแก้ไข Bitcoin จึงได้มีการทำ OFF-CHAIN Solution ขึ้นมาเพื่อแก้ปัญหา Transaction ที่แออัด เช่น Lightning Network ที่จะช่วยลดความแออัดของธุรกรรมบน Network หลักของ Bitcoin และนอกจากนั้นการทำ Private Blockchain Framework ก็สามารถช่วยในเรื่องการลด Traffic ที่เข้าใน Network ได้



Reference:

https://www.researchgate.net/figure/Cryptocurrencies-transaction-speeds-compared-to-Visa-and-Paypal-74_fig2_338792619

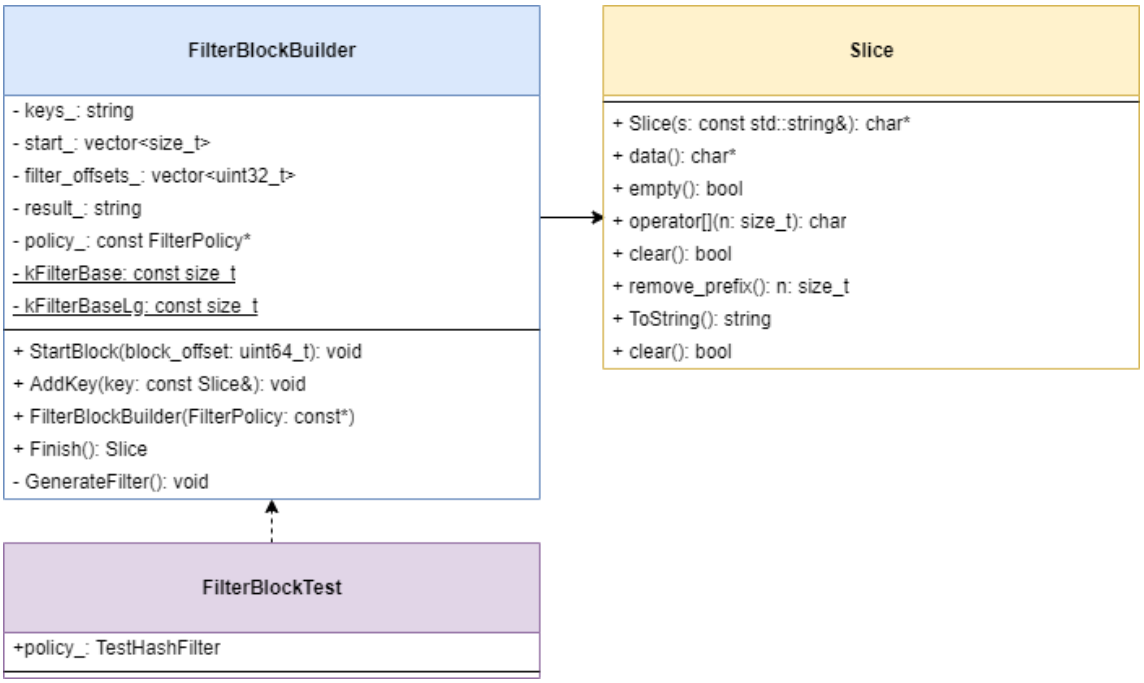
Design Pattern ของ Bitcoin

1. Builder

เนื่องจาก Bitcoin เป็น Cryptocurrency ที่ใช้เทคโนโลยี Blockchain เข้ามาทำธุรกรรมการเงิน ซึ่ง Builder Design Pattern มีส่วนในการช่วยให้มีการสร้าง Block เหล่านี้ได้ต่อๆกัน

โดยจากตัวอย่างโค้ด จะเห็นได้ว่า ระหว่างการสร้างแต่ละฟิลเตอร์บล็อก จะใช้ Filter BlockBuilder เพื่อสร้าง filter key ต่างๆ ขึ้นมาเพื่อคัดกรองบล็อกที่จะถูกส่งเข้าไปในเครือข่าย โดยมีตัวแปรที่เรากำหนดไว้เพื่อใช้ในการคัดกรองบล็อกต่างๆ และทันทีที่บล็อกนั้นสร้างสำเร็จ FilterBlock Builder ก็จะหยุดทำงาน เป็นอันเสร็จสิ้นการสร้าง 1 ฟิลเตอร์บล็อกของ Bitcoin โดยเมื่อมีการสร้าง 1 ฟิลเตอร์บล็อกเสร็จแล้ว ก็จะถูกเก็บอยู่ในคลาส Slice เพื่อเก็บเป็น Data ให้ FilterBlockReader ในขั้นต่อไปนำข้อมูล Data ที่ฟิลเตอร์มาเสร็จเรียบร้อยไปประมวลผลต่อไป

UML Class Diagram



filter_block.h (24-102)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/table/filter_block.h

filter_block_test.cc (53-72)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/table/filter_block_test.cc

slice.h (28-92)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/include/leveldb/slice.h

Source Code

```
class FilterBlockBuilder {
 public:
  explicit FilterBlockBuilder(const FilterPolicy*);

  FilterBlockBuilder(const FilterBlockBuilder&) = delete;
  FilterBlockBuilder& operator=(const FilterBlockBuilder&) = delete;

  void StartBlock(uint64_t block_offset);
  void AddKey(const Slice& key);
  Slice Finish();

 private:
  void GenerateFilter();

  const FilterPolicy* policy_;
  std::string keys_;              // Flattened key contents
  std::vector<size_t> start_;     // Starting index in keys_ of each key
  std::string result_;            // Filter data computed so far
  std::vector<Slice> tmp_keys_;  // policy_->CreateFilter() argument
  std::vector<uint32_t> filter_offsets_;
};
```

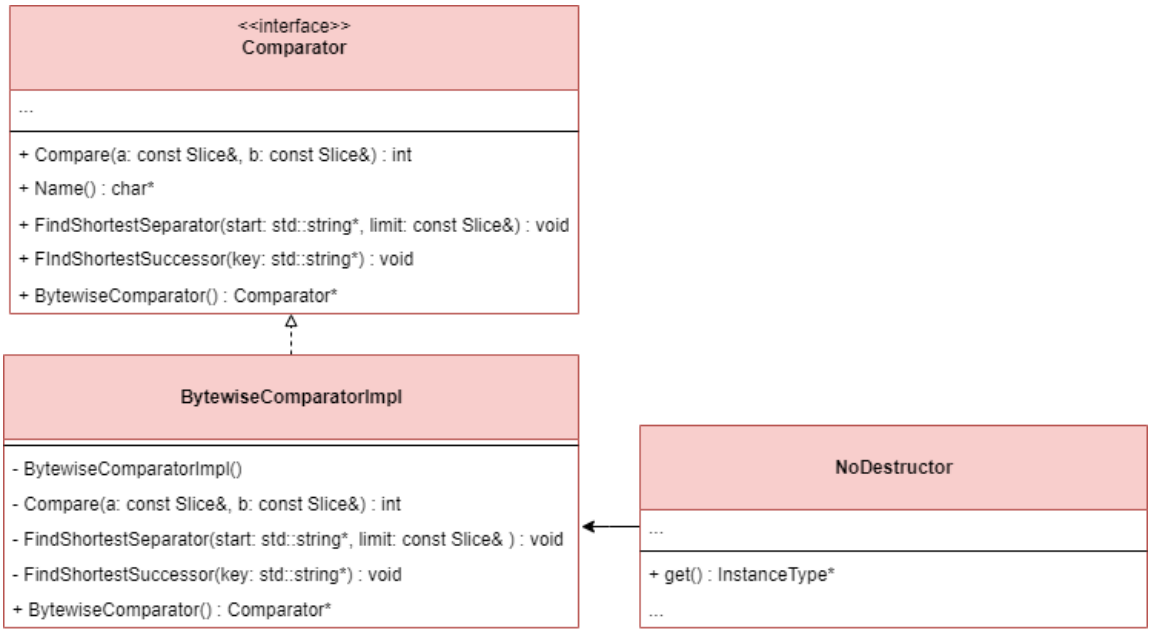
```
TEST(FilterBlockTest, SingleChunk) {
  FilterBlockBuilder builder(&policy_);
  builder.StartBlock(100);
  builder.AddKey("foo");
  builder.AddKey("bar");
  builder.AddKey("box");
  builder.StartBlock(200);
  builder.AddKey("box");
  builder.StartBlock(300);
  builder.AddKey("hello");
  Slice block = builder.Finish();
  FilterBlockReader reader(&policy_, block);
  ASSERT_TRUE(reader.KeyMayMatch(100, "foo"));
  ASSERT_TRUE(reader.KeyMayMatch(100, "bar"));
  ASSERT_TRUE(reader.KeyMayMatch(100, "box"));
  ASSERT_TRUE(reader.KeyMayMatch(100, "hello"));
  ASSERT_TRUE(reader.KeyMayMatch(100, "foo"));
  ASSERT_TRUE(!reader.KeyMayMatch(100, "missing"));
  ASSERT_TRUE(!reader.KeyMayMatch(100, "other"));
}
```

2. Singleton

ใน Bitcoin มีข้อมูลจำนวนมหาศาลที่ใช้สื่อสารกันระหว่าง Node นับล้านๆ แน่นอนว่าส่วนสำคัญที่สุดของเครือข่าย Blockchain คือการเปรียบเทียบข้อมูลของแต่ละ Node ว่ามีข้อมูลที่ตรงกันหรือไม่ โดย ณ ที่นี้จะใช้คลาส Interface Comparator เป็น Interface แม่แบบเพื่อให้คลาสอื่นๆสามารถนำไปใช้เปรียบเทียบข้อมูลที่ส่วนนั้นๆรับผิดชอบได้ โดยส่วนถูกเรียกใช้งานมากที่สุดคือการเปรียบเทียบระหว่าง Byte ของข้อมูล ทำให้เกิดคลาส BytewiseComparator โดยมี คลาส BytewiseComparatorImpl เป็นแม่แบบให้ใช้งาน แน่นอนว่าในการเปรียบเทียบข้อมูลระดับ Byte นั้นจะเกิดขึ้นเป็นจำนวนมากมายนับไม่ถ้วน และจะถูกเปรียบเทียบในหลายๆส่วนของโค้ด

จึงทำให้การเรียกใช้ BytewiseComparator เกิดขึ้นมากมาย ดังนั้นการใช้ Singleton Design Pattern จะทำให้ไม่เปลืองทรัพยากรในการสร้าง Instance ของ BytewiseComparator Impl และยังทำให้สามารถใช้งาน BytewiseComparator ได้จากที่เดียวกัน นี่คือเหตุผลว่าทำไมรูปแบบSingleton จึงเหมาะสมกับ คลาส BytewiseComparatorImpl

UML Class Diagram



comparator.cc (70-73)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/util/comparator.cc

no_destructor.h (17-42)

bitcoin/no_destructor.h at 7fcf53f7b4524572d1d0c9a5fdc388e87eb02416 · bitcoin/bitcoin (github.com)

comparator.h (20-55)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/include/leveldb/comparator.h

Source Code

```
namespace leveldb {

Comparator::~Comparator() = default;

namespace {
class BytewiseComparatorImpl : public Comparator {
 public:
  BytewiseComparatorImpl() = default;

  const char* Name() const override { return "leveldb.BytewiseComparator"; }

  int Compare(const Slice& a, const Slice& b) const override {
    return a.compare(b);
  }

  void FindShortestSeparator(std::string* start,
                             const Slice& limit) const override {
    // Find length of common prefix
    size_t min_length = std::min(start->size(), limit.size());
    size_t diff_index = 0;
    while ((diff_index < min_length) &&
           ((*start)[diff_index] == limit[diff_index])) {
      diff_index++;
    }
```

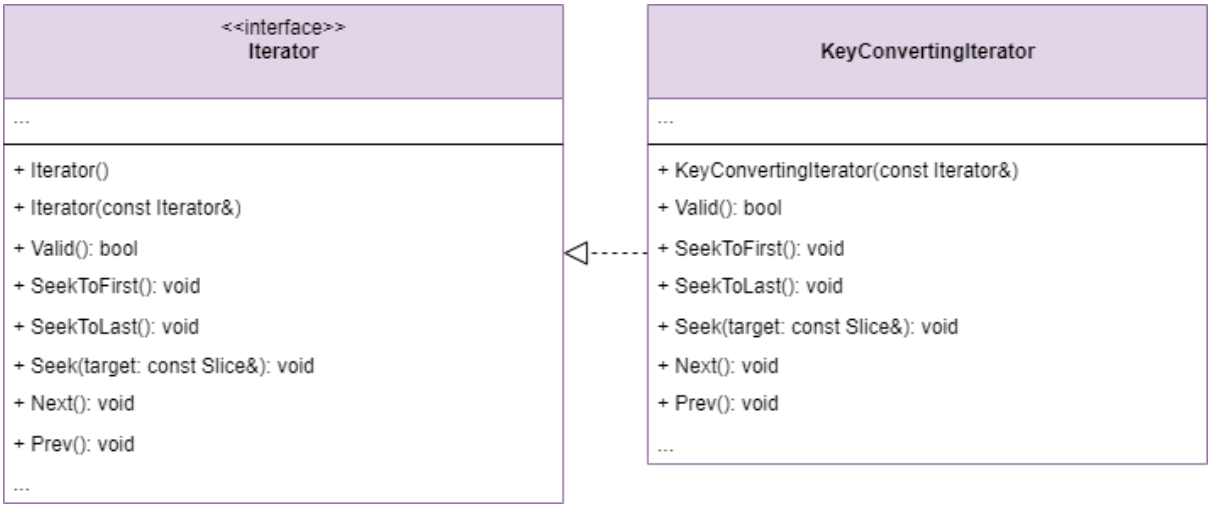
```
  const Comparator* BytewiseComparator() {
    static NoDestructor<BytewiseComparatorImpl> singleton;
    return singleton.get();
  }

}  // namespace leveldb
```

```
namespace leveldb {

// Wraps an instance whose destructor is never called.
//
// This is intended for use with function-level static variables.
template <typename InstanceType>
class NoDestructor {
 public:
  template <typename... ConstructorArgTypes>
  explicit NoDestructor(ConstructorArgTypes&&... constructor_args) {
    static_assert(sizeof(instance_storage_) >= sizeof(InstanceType),
                  "instance_storage_ is not large enough to hold the instance");
    static_assert(
        alignof(decltype(instance_storage_)) >= alignof(InstanceType),
        "instance_storage_ does not meet the instance's alignment requirement");
    new (&instance_storage_)
        InstanceType(std::forward<ConstructorArgTypes>(constructor_args)...);
  }

  ~NoDestructor() = default;

  NoDestructor(const NoDestructor&) = delete;
  NoDestructor& operator=(const NoDestructor&) = delete;

  InstanceType* get() {
    return reinterpret_cast<InstanceType*>(&instance_storage_);
  }

 private:
  typename std::aligned_storage<sizeof(InstanceType),
                                alignof(InstanceType)>::type instance_storage_;
};

}  // namespace leveldb
```

3. Iterator

เนื่องจากใน Bitcoin มีข้อมูลจำนวนมหาศาลที่ใช้สื่อสารกันระหว่าง Node นับล้านๆ แน่นอนว่าส่วนสำคัญที่สุดของเครือข่าย Blockchain คือการเข้าถึงข้อมูลเพื่อที่จะตรวจสอบข้อมูลต่างๆ โดย ณ ทีนี้จะใช้คลาส Interface Iterator เป็น Interface แม่แบบเพื่อให้คลาสอื่นๆสามารถนำไปใช้เข้าถึงข้อมูลที่ส่วนนั้นๆรับผิดชอบได้ ส่วนสำคัญคือในการเข้าถึงข้อมูลจำนวนมาก ระบบจะต้องพบกับข้อมูลหลากหลายรูปแบบ หลากหลายชนิด การใช้ Iterator Pattern จึงเป็นการกำหนดรูปแบบการเข้าถึงข้อมูล ให้มีความหมายหรือความเข้าใจตรงกัน

UML Diagram



iterator.h (24-81)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/include/leveldb/iterator.h

table_test.cc (261 - 300)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/table/table_test.cc

Source Code

```cpp
class LEVELDB_EXPORT Iterator {
 public:
  Iterator();

  Iterator(const Iterator&) = delete;
  Iterator& operator=(const Iterator&) = delete;

  virtual ~Iterator();
  virtual bool Valid() const = 0;
  virtual void SeekToFirst() = 0;
  virtual void SeekToLast() = 0;
  virtual void Seek(const Slice& target) = 0;
  virtual void Next() = 0;
  virtual void Prev() = 0;
  virtual Slice key() const = 0;
  virtual Slice value() const = 0;
  virtual Status status() const = 0;
  using CleanupFunction = void (*)(void* arg1, void* arg2);
  void RegisterCleanup(CleanupFunction function, void* arg1, void* arg2);
```

```cpp
class KeyConvertingIterator : public Iterator {
 public:
  explicit KeyConvertingIterator(Iterator* iter) : iter_(iter) {}

  KeyConvertingIterator(const KeyConvertingIterator&) = delete;
  KeyConvertingIterator& operator=(const KeyConvertingIterator&) = delete;

  ~KeyConvertingIterator() override { delete iter_; }

  bool Valid() const override { return iter_->Valid(); }
  void Seek(const Slice& target) override {
    ParsedInternalKey ikey(target, kMaxSequenceNumber, kTypeValue);
    std::string encoded;
    AppendInternalKey(&encoded, ikey);
    iter_->Seek(encoded);
  }
  void SeekToFirst() override { iter_->SeekToFirst(); }
  void SeekToLast() override { iter_->SeekToLast(); }
  void Next() override { iter_->Next(); }
  void Prev() override { iter_->Prev(); }

  Slice key() const override {
    assert(Valid());
    ParsedInternalKey key;
    if (!ParseInternalKey(iter_->key(), &key)) {
      status_ = Status::Corruption("malformed internal key");
      return Slice("corrupted key");
    }
    return key.user_key;
  }

  Slice value() const override { return iter_->value(); }
  Status status() const override {
    return status_.ok() ? iter_->status() : status_;
  }

 private:
  mutable Status status_;
  Iterator* iter_;
};
```

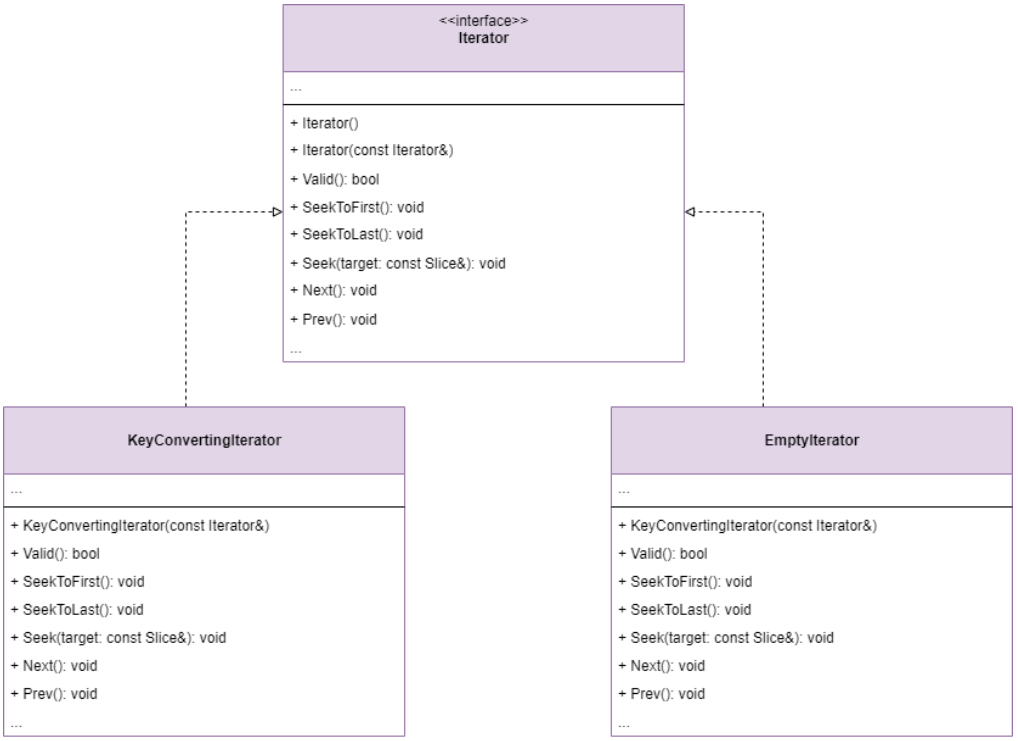
4. Null Object

ใน Bitcoin มีข้อมูลจำนวนมหาศาลที่ใช้สื่อสารกันระหว่าง Node นับล้านๆ แน่นอนว่าส่วนสำคัญที่สุดของเครือข่าย Blockchain คือการเข้าถึงข้อมูลเพื่อที่จะตรวจสอบข้อมูลต่างๆ

โดย ณ ทีนี้จะใช้คลาส Interface Iterator เป็น Interface แม่แบบเพื่อให้คลาสอื่นๆสามารถนำไปใช้เข้าถึงข้อมูลที่ส่วนนั้นๆรับผิดชอบได้ ส่วนสำคัญคือในการเข้าถึงข้อมูลจำนวนมาก ระบบจะต้องพบกับข้อมูลหลากหลายรูปแบบ หลากหลายชนิด ซึ่งนั่นหมายถึงมีโอกาสที่หนึ่งในคลาสที่สืบทอด Iterator จะได้พบกับข้อมูลประเภท null ย่อมเกิดขึ้นได้

นี่คือส่วนสำคัญที่ Bitcoin จะต้องมีรูปแบบ Null Object เพื่อป้องกันข้อผิดพลาดที่อาจเกิดขึ้นได้จากข้อมูลประเภท null ซึ่งถ้าหากเกิดข้อผิดพลาดขึ้นแม้เพียงจุดเล็กๆอาจทำให้เครือข่ายทั้งเครือข่ายใช้งานไม่ได้จนเกิดความเสียหายในวงกว้าง จึงต้องมีรูปแบบ Null Object เพื่อป้องกันข้อผิดพลาดที่อาจเกิดขึ้นในส่วนนี้

UML Class Diagram



iterator.h (24-81)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/include/leveldb/iterator.h

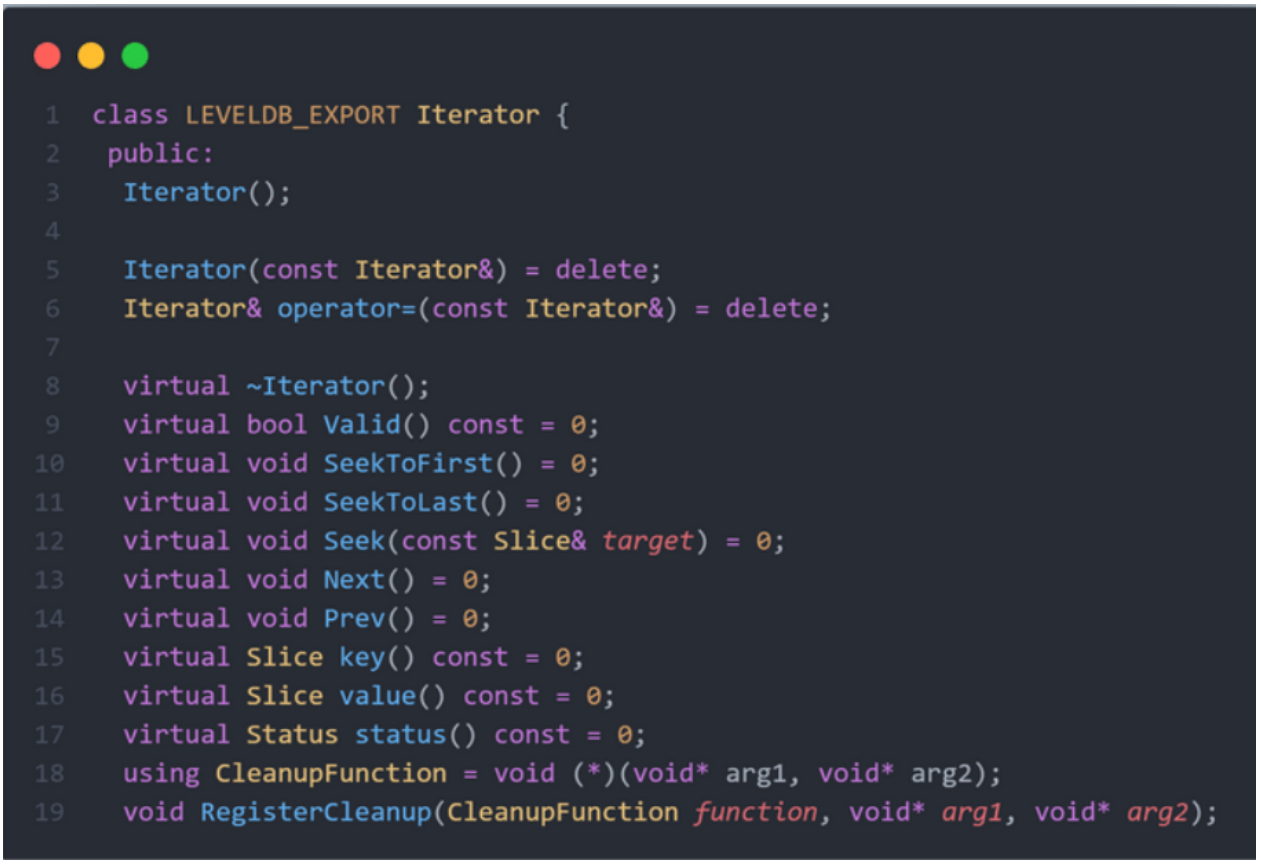
iterator.cc (43-66)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/table/iterator.cc

table_test.cc (261 - 300)

https://github.com/bitcoin/bitcoin/blob/7fcf53f7b4524572d1d0c9a5fdc388e87eb02416/src/leveldb/table/table_test.cc

Source Code



```
class LEVELDB_EXPORT Iterator {
 public:
  Iterator();

  Iterator(const Iterator&) = delete;
  Iterator& operator=(const Iterator&) = delete;

  virtual ~Iterator();
  virtual bool Valid() const = 0;
  virtual void SeekToFirst() = 0;
  virtual void SeekToLast() = 0;
  virtual void Seek(const Slice& target) = 0;
  virtual void Next() = 0;
  virtual void Prev() = 0;
  virtual Slice key() const = 0;
  virtual Slice value() const = 0;
  virtual Status status() const = 0;
  using CleanupFunction = void (*)(void* arg1, void* arg2);
  void RegisterCleanup(CleanupFunction function, void* arg1, void* arg2);
```

```
class EmptyIterator : public Iterator {
 public:
  EmptyIterator(const Status& s) : status_(s) {}
  ~EmptyIterator() override = default;

  bool Valid() const override { return false; }
  void Seek(const Slice& target) override {}
  void SeekToFirst() override {}
  void SeekToLast() override {}
  void Next() override { assert(false); }
  void Prev() override { assert(false); }
  Slice key() const override {
    assert(false);
    return Slice();
  }
  Slice value() const override {
    assert(false);
    return Slice();
  }
  Status status() const override { return status_; }

 private:
  Status status_;
};
```

```
class KeyConvertingIterator : public Iterator {
 public:
  explicit KeyConvertingIterator(Iterator* iter) : iter_(iter) {}

  KeyConvertingIterator(const KeyConvertingIterator&) = delete;
  KeyConvertingIterator& operator=(const KeyConvertingIterator&) = delete;

  ~KeyConvertingIterator() override { delete iter_; }

  bool Valid() const override { return iter_->Valid(); }
  void Seek(const Slice& target) override {
    ParsedInternalKey ikey(target, kMaxSequenceNumber, kTypeValue);
    std::string encoded;
    AppendInternalKey(&encoded, ikey);
    iter_->Seek(encoded);
  }
  void SeekToFirst() override { iter_->SeekToFirst(); }
  void SeekToLast() override { iter_->SeekToLast(); }
  void Next() override { iter_->Next(); }
  void Prev() override { iter_->Prev(); }

  Slice key() const override {
    assert(Valid());
    ParsedInternalKey key;
    if (!ParseInternalKey(iter_->key(), &key)) {
      status_ = Status::Corruption("malformed internal key");
      return Slice("corrupted key");
    }
    return key.user_key;
  }

  Slice value() const override { return iter_->value(); }
  Status status() const override {
    return status_.ok() ? iter_->status() : status_;
  }

 private:
  mutable Status status_;
  Iterator* iter_;
};
```