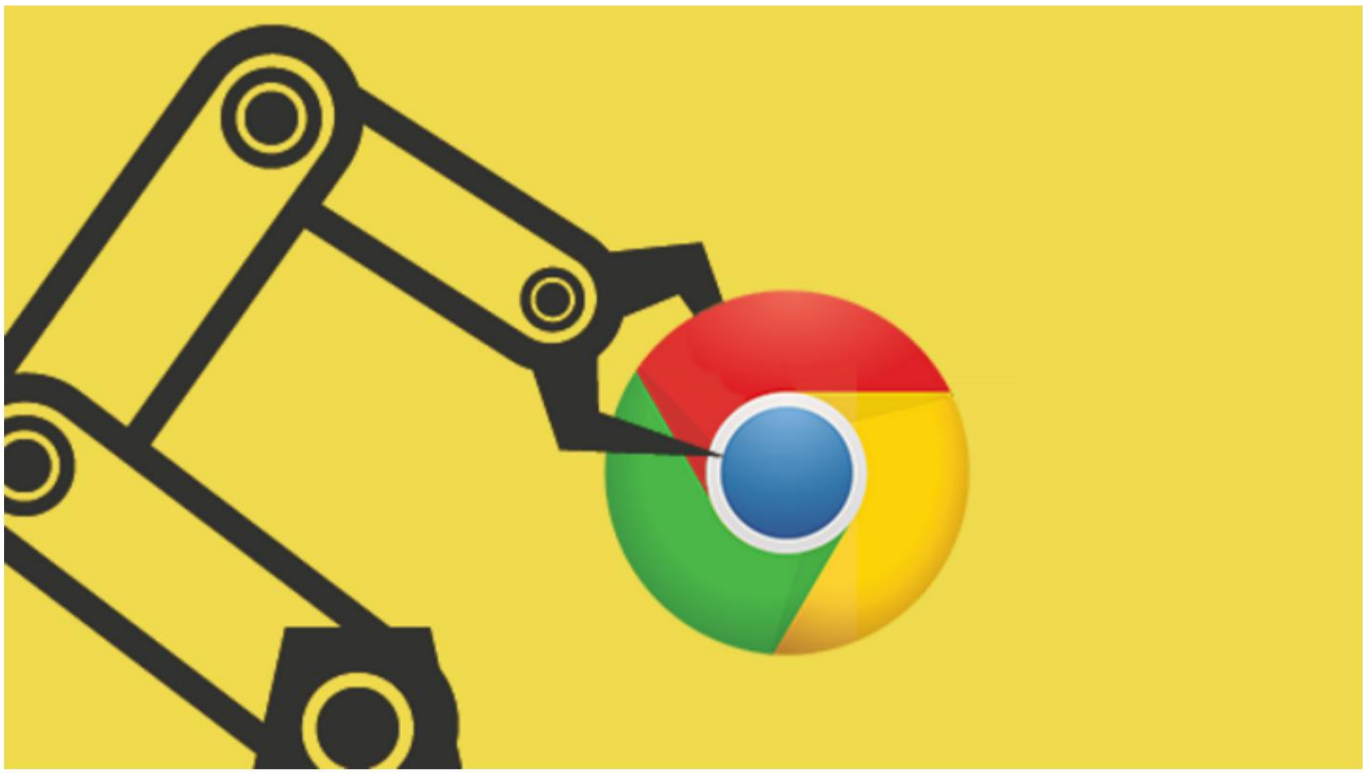
"Chrome V8 Source Code" 27. The mysterious and simple dispatch_table_



1 abstract

This article is the third of the Builtin topic, explaining the execution of Bytecode, data structure and Dispatch. dispatch_table_is the connection
The link between Bytecodes, which records the address of each Bytecode handler, Ignition searches and executes the corresponding
Bytecode. The content of this article is organized as follows: Bytecode execution and data structure (Chapter 2); Bytecode scheduling (Chapter 3).

2 Bytecode execution

In V8, the interpreter responsible for executing Bytecode is Ignition. When Ignition executes Bytecode, it needs to make a lot of complicated preparations. These "preparations"
Work" follow-up article explains, we focus on the execution of Bytecode.

Bytecode is generated at the granularity of JavaScript functions and stored in Bytecode array, that is, Bytecode array is an array that stores Bytecode.

The source code is as follows:

```
1. class BytecodeArray : public FixedArrayBase {
2. public:
3.         static constexpr int SizeFor(int length) {
4.             return OBJECT_POINTER_ALIGN(kHeaderSize + length);
5.         }
6.         inline byte get(int index) const;
7.         inline void set(int index, byte value);
8.         inline Address GetFirstBytecodeAddress();
9. //Omit.....
10. };
```

We only explain two points relevant to this article:

(1) Line 3 of the code SizeFor(int length) calculates the length of the Bytecode array. The value of the parameter length is after compiling the JavaScript function.
The number of Bytecodes obtained, the space required by length+Bytecode array is equal to the length of Bytecode array. When creating Bytecode array
When, use SizeFor(int length) to calculate the length of the requested memory;

(2) Line 8 of code GetFirstBytecodeAddress() obtains the first address of Bytecode. Copy the Bytecode generated by Parser to
This function is used when Bytecode array.
In Factory::NewBytecodeArray(), use the return value of SizeFor(int length) to apply for memory, and use CopyBytes() to
Bytecode is copied to the first address. The following is a piece of Bytecode source code:

```
1. a7                   StackCheck
2.12 00                 LdaConstant [0]
3. 15 01 00             StaGlobal [1], [0]
4. 13 01 02             LdaGlobal [1], [2]
5. 26 f9                Star r2
6. 29 f9 02             LdaNamedPropertyNoFeedback r2, [2]
7. 26fa                 Star r1
8. 0c 05                LdaSmi [5]
9. Constant pool (size = 6)
10. 0000005EAE403019: [FixedArray] in OldSpace
11.      - map: 0x03d0be000169 <Map>
12.      - length: 6
13.                  0: 0x005eae402f59 <String[#22]: ignoreCase here we go!>
14.                  1: 0x038ee90c3cc1 <String[#1]: a>
15.                  2: 0x01b92bc2bde1 <String[#9]: substring>
16.                  3: 0x038ee90c3fa9 <String[#1]: b>
17.                  4: 0x01b92bc33839 <String[#7]: console>
18.                  5: 0x01b92bc32e79 <String[#3]: log>
```

Line 2 of code 12 00 LdaConstant [0]: 12 is the number of LdaConstant, which is also the enumeration value of LdaConstant, that is
Bytecode[0x12]=kLdaConstant, the source code is as follows:

```
enum class Bytecode : uint8_t {

            kWide, kExtraWide, kDebugBreakWide, kDebugBreakExtraWide, kDebugBreak0,
kDebugBreak1, kDebugBreak2, kDebugBreak3, kDebugBreak4, kDebugBreak5,
kDebugBreak6, kLdaZero, kLdaSmi, kLdaUndefined, kLdaNull, kLdaTheHole, kLdaTrue,
kLdaFalse, kLdaConstant, kLdaGlobal, kLdaGlobalInsideTypeof, kStaGlobal,
kPushContext, kPopContext, kLdaContextSlot, kLdaImmutableContextSlot,
kLdaCurrentContextSlot, kLdaImmutableCurrentContextSlot, kStaContextSlot,
kStaCurrentContextSlot, kLdaLookupSlot, kLdaLookupContextSlot,
kLdaLookupGlobalSlot, kLdaLookupSlotInsideTypeof,
kLdaLookupContextSlotInsideTypeof, kLdaLookupGlobalSlotInsideTypeof,
kStaLookupSlot, kLdar, kStar, kMov, kLdaNamedProperty,
kLdaNamedPropertyNoFeedback, kLdaKeyedProperty, kLdaModuleVariable,
kStaModuleVariable, kStaNamedProperty, kStaNamedPropertyNoFeedback,
kStaNamedOwnProperty, kStaKeyedProperty, kStaInArrayLiteral,
kStaDataPropertyInLiteral, kCollectTypeProfile, kAdd, kSub, kMul, kDiv, kMod,
kExp, kBitwiseOr, kBitwiseXor, kBitwiseAnd, kShiftLeft, kShiftRight,
```

```
kShiftRightLogical, kAddSmi, kSubSmi, kMulSmi, kDivSmi, kModSmi, kExpSmi,
kBitwiseOrSmi, kBitwiseXorSmi, kBitwiseAndSmi, kShiftLeftSmi, kShiftRightSmi,
kShiftRightLogicalSmi, kInc, kDec, kNegate, kBitwiseNot, kToBooleanLogicalNot,
kLogicalNot, kTypeOf, kDeletePropertyStrict, //Omit.....
}
```

V8 stipulates: fb represents register R0, fa represents register R1, and so on. At 29 f9 02 LdaNamedPropertyNoFeedback r2,
In [2], f9 represents register R2, and 02 represents the constant pool [2]. When executing LdaNamedPropertyNoFeedback, Ignition obtains
Get the base address of dispatch_table, and then get the LdaNamedPropertyNoFeedback through base address+0x29
handler, the source code is as follows:

```
// Calls the GetProperty builtin for <object> and the key in the accumulator.
IGNITION_HANDLER(LdaNamedPropertyNoFeedback, InterpreterAssembler) {
    TNode<Object> object = LoadRegisterAtOperandIndex(0);
    TNode<Name> name = CAST(LoadConstantPoolEntryAtOperandIndex(1));
    TNode<Context> context = GetContext();
    TNode<Object> result =
        CallBuiltin(Builtins::kGetProperty, context, object, name);
    SetAccumulator(result);
    Dispatch();
}
```

3 Dispatch

Dispatch_table is an array of pointers. The enumeration value of Bytecode represents its position in the array, and the corresponding Bytecode is stored in this position.
The address of the handler. The initialization of Dispatch_table is as follows:

```
1. void Interpreter::Initialize() {
2.     Builtins* builtins = isolate_->builtins();
3.     // Set the interpreter entry trampoline entry point now that builtins are
4.     // initialized.
5.     Handle<Code> code = BUILTIN_CODE(isolate_, InterpreterEntryTrampoline);
6.     DCHECK(builtins->is_initialized());
7.     DCHECK(code->is_off_heap_trampoline() ||
8.            isolate_->heap()->IsImmovable(*code));
9.     interpreter_entry_trampoline_instruction_start_ = code->InstructionStart();
10.    // Initialize the dispatch table.
11.    Code illegal = builtins->builtin(Builtins::kIllegalHandler);
12.    int builtin_id = Builtins::kFirstBytecodeHandler;
13.    ForEachBytecode([=, &builtin_id](Bytecode bytecode,
14.                                     OperandScale operand_scale) {
15.      Code handler = illegal;
16.      if (Bytecodes::BytecodeHasHandler(bytecode, operand_scale)) {
17. #ifdef DEBUG
18.        std::string builtin_name(Builtins::name(builtin_id));
19.        std::string expected_name =
20.            Bytecodes::ToString(bytecode, operand_scale, "") + "Handler";
```

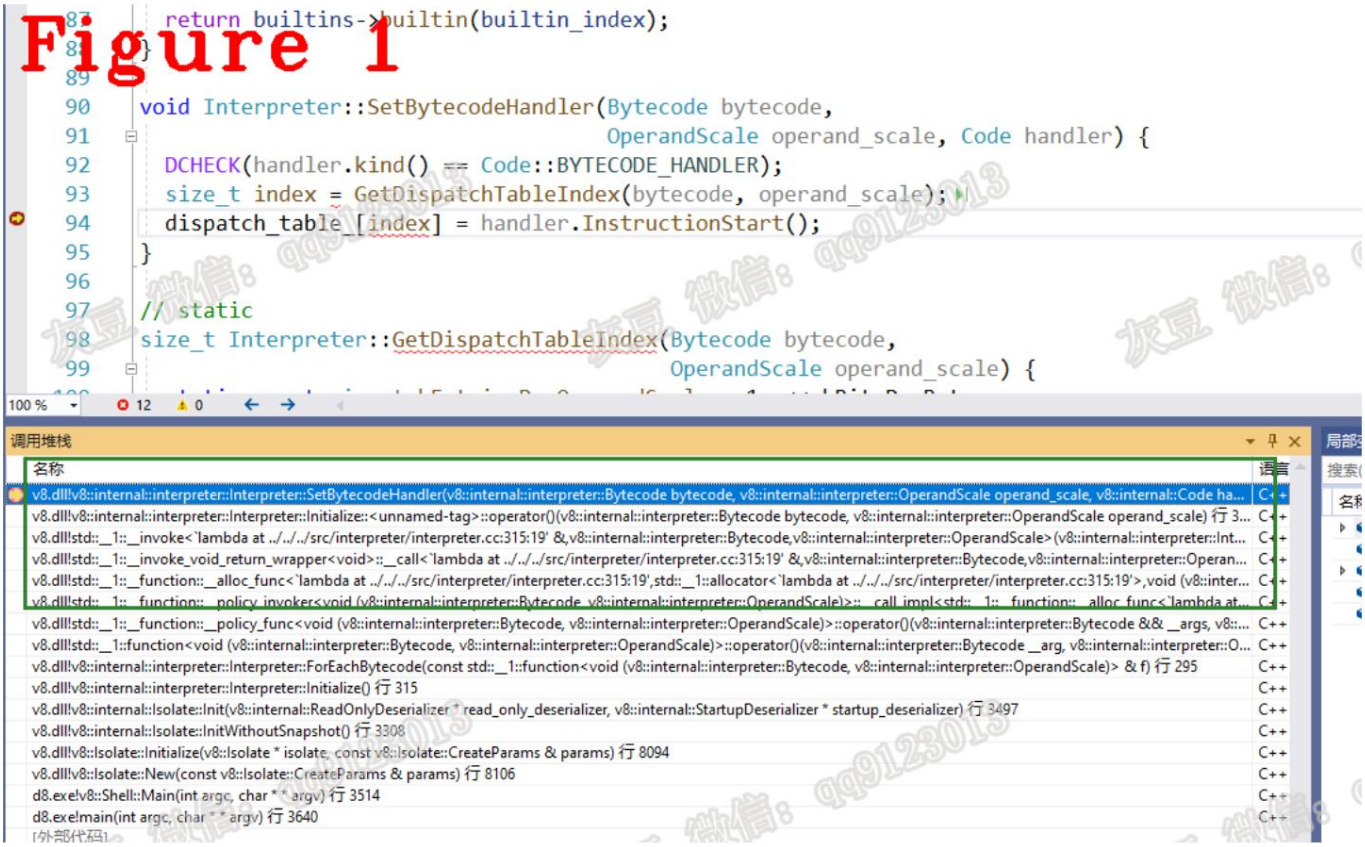
```
                DCHECK_EQ(expected_name, builtin_name);
#endif
                handler = builtins->builtin(builtin_id++);
            }
            SetBytecodeHandler(bytecode, operand_scale, handler);
        });
        DCHECK(builtin_id == Builtins::builtin_count);
        DCHECK(IsDispatchTableInitialized());
}
```

The above 13-26 lines of code are anonymous functions, of which 25 lines of code initialize Dispatch_table. The source code is as follows:

```
void Interpreter::SetBytecodeHandler(Bytecode bytecode,
                                     OperandScale operand_scale, Code handler)
{
    DCHECK(handler.kind() == Code::BYTECODE_HANDLER);
    size_t index = GetDispatchTableIndex(bytecode, operand_scale);
    dispatch_table_[index] = handler.InstructionStart();
}
//.....Separator line..... .....
size_t Interpreter::GetDispatchTableIndex(Bytecode bytecode,
                                          OperandScale operand_scale) {
    static const size_t kEntriesPerOperandScale = 1u << kBitsPerByte;
    size_t index = static_cast<size_t>(bytecode);
    return index + BytecodeOperands::OperandScaleAsIndex(operand_scale) *
                   kEntriesPerOperandScale;
}
```

The above 5th line of code dispatch_table_ is the member variable that stores the dispatch table that we have been thinking about for a long time; the 4th line of code GetDispatchTableIndex() calculates the position of the Bytecode handler in dispatch_table. This position is consistent with the enum class Bytecode is the same. Figure 1 shows the call stack of SetBytecodeHandler.

Figure 1



The source code of Interpreter is as follows:

```
1. class Interpreter {
2.          //.....omitted.....
3.private :
4.       // Get dispatch table index of bytecode.
5.       static size_t GetDispatchTableIndex(Bytecode bytecode,
6.                                           OperandScale operand_scale);
7.       static const int kNumberOfWideVariants =
BytecodeOperands::kOperandScaleCount;
8.       static const int kDispatchTableSize = kNumberOfWideVariants * (kMaxUInt8 +
1);
9.       static const int kNumberOfBytecodes = static_cast<int>(Bytecode::kLast) + 1;
10.       Isolate* isolate_;
11.       Address dispatch_table_[kDispatchTableSize];
12.       std::unique_ptr<uintptr_t[]> bytecode_dispatch_counters_table_;
13.       Address interpreter_entry_trampoline_instruction_start_;
14.       DISALLOW_COPY_AND_ASSIGN(Interpreter);
15. };
```

The dispatch_table_ in the 11th line of code above is a member variable of Interpreter. Interpreter is a member variable of Isolate, the source code is as follows

Down:

```
1. class Isolate final : private HiddenFactory {
2. //Omit.....
3.       const AstStringConstants* ast_string_constants_ = nullptr;
4.       interpreter::Interpreter* interpreter_ = nullptr;
5.       compiler::PerIsolateCompilerCache* compiler_cache_ = nullptr;
```

```
6.      Zone* compiler_zone_ = nullptr;
7.      CompilerDispatcher* compiler_dispatcher_ = nullptr;
8.      friend class heap::HeapTester;
9.      friend class TestSerializer;
10.       DISALLOW_COPY_AND_ASSIGN(Isolate);
11. };
```

It can be seen from the above code: Isolate->interpreter_->dispatch_table_gets dispatch_table_. Below is the Source code of Dispatch() called in Bytecode handler:

```
1. void InterpreterAssembler::Dispatch() {
2.      Comment("========== Dispatch");
3.      DCHECK_IMPLIES(Bytecodes::MakesCallAlongCriticalPath(bytecode_),
made_call_);
4.      TNode<IntPtrT> target_offset = Advance();
5.      TNode<WordT> target_bytecode = LoadBytecode(target_offset);
6.      if (Bytecodes::IsStarLookahead(bytecode_, operand_scale_)) {
7.        target_bytecode = StarDispatchLookahead(target_bytecode);
8.      }
9.      DispatchToBytecode(target_bytecode, BytecodeOffset());
10. }
11. //......Divider line.....................................
12. void InterpreterAssembler::DispatchToBytecode(
13.            TNode<WordT> target_bytecode, TNode<IntPtrT> new_bytecode_offset) {
14. if (FLAG_trace_ignition_dispatches) {
15.            TraceBytecodeDispatch(target_bytecode);
16.        }
17.        TNode<RawPtrT> target_code_entry = Load<RawPtrT>(
18.            DispatchTablePointer(), TimesSystemPointerSize(target_bytecode));
19.        DispatchToBytecodeHandlerEntry(target_code_entry, new_bytecode_offset);
20.     }
21. //.............Separation line..... .....
22. void InterpreterAssembler::DispatchToBytecodeHandlerEntry(
twenty three.           TNode<RawPtrT> handler_entry, TNode<IntPtrT> bytecode_offset) {
twenty four.        // Propagate speculation poisoning.
25.        TNode<RawPtrT> poisoned_handler_entry =
26.            UncheckedCast<RawPtrT>(WordPoisonOnSpeculation(handler_entry));
27.        TailCallBytecodeDispatch(InterpreterDispatchDescriptor{},
28.                                 poisoned_handler_entry,
GetAccumulatorUnchecked(),
29.                                 bytecode_offset, BytecodeArrayTaggedPointer(),
30.                                 DispatchTablePointer());
31.     }
32.       //.............Separator line..... .
33. void CodeAssembler::TailCallBytecodeDispatch(
34. const CallInterfaceDescriptor& descriptor, TNode<RawPtrT> target,
35.          TArgs... args) {
36.      DCHECK_EQ(descriptor.GetParameterCount(), sizeof...(args));
37.      auto call_descriptor = Linkage::GetBytecodeDispatchCallDescriptor(
38.          zone(), descriptor, descriptor.GetStackParameterCount());
39.      Node* nodes[] = {target, args...};
```

```
40.     CHECK_EQ(descriptor.GetParameterCount() + 1, arraysize(nodes)); raw_assembler()-
41.     >TailCallN(call_descriptor, arraysize(nodes), nodes);
42. }
```

The above three methods jointly implement Bytecode dispatch. The 5th line of code calculates target_bytecode; the 17th line of code calculates target_bytecode_entry; the 27th line of code starts to jump; the 34th line of code creates a call discriptor; the 41st line of code generates a Node node and adds the node to the end of the current basic block , the jump is completed. For a detailed explanation of TailCallN(), see the eleventh article. Figure 2 shows the call stack of Dispatch().

Figure 2

```
1327    DispatchToBytecodeHandlerEntry(target_code_entry, new_bytecode_offset);
1328  }
1329
1330  void InterpreterAssembler::DispatchToBytecodeHandlerEntry(
1331      TNode<RawPtrT> handler_entry, TNode<IntPtrT> bytecode_offset) {
1332    // Propagate speculation poisoning.
1333    TNode<RawPtrT> poisoned_handler_entry =
1334        UncheckedCast<RawPtrT>(WordPoisonOnSpeculation(handler_entry));
1335    TailCallBytecodeDispatch(InterpreterDispatchDescriptor{},
1336                             poisoned_handler_entry, GetAccumulatorUnchecked(),
1337                             bytecode_offset, BytecodeArrayTaggedPointer(),
1338                             DispatchTablePointer());
1339  }
1340
1341  void InterpreterAssembler::DispatchWide(OperandScale operand_scale) {
1342    // Dispatching a wide bytecode requires treating the prefix
1343    // bytecode a base pointer into the dispatch table and dispatching
1344    // the bytecode that follows relative to this base.
```

100 % 38 0

调用堆栈

名称	语
v8.dll!v8::internal::interpreter::InterpreterAssembler::DispatchToBytecodeHandlerEntry(v8::internal::compiler::TNode<v8::internal::RawPtrT> handler_entry, v8::internal::compiler::TNod...	C-
v8.dll!v8::internal::interpreter::InterpreterAssembler::DispatchWide(v8::internal::interpreter::OperandScale operand_scale) 行 1372	C-
v8.dll!v8::internal::interpreter::`anonymous namespace'::WideAssembler::GenerateImpl() 行 3227	C-
v8.dll!v8::internal::interpreter::`anonymous namespace'::WideAssembler::Generate(v8::internal::compiler::CodeAssemblerState * state, v8::internal::interpreter::OperandScale scale) 行 ...	C-
v8.dll!v8::internal::interpreter::GenerateBytecodeHandler(v8::internal::Isolate * isolate, const char * debug_name, v8::internal::interpreter::Bytecode bytecode, v8::internal::interpreter::O...	C-
v8.dll!v8::internal::`anonymous namespace'::GenerateBytecodeHandler(v8::internal::Isolate * isolate, int builtin_index, v8::internal::interpreter::OperandScale operand_scale, v8::internal...	C-
v8.dll!v8::internal::SetupIsolateDelegate::SetupBuiltinsInternal(v8::internal::Isolate * isolate) 行 325	C-
v8.dll!v8::internal::SetupIsolateDelegate::SetupBuiltins(v8::internal::Isolate * isolate) 行 20	C-
v8.dll!v8::internal::Isolate::Init(v8::internal::ReadOnlyDeserializer * read_only_deserializer, v8::internal::StartupDeserializer * startup_deserializer) 行 3445	C-
v8.dll!v8::internal::Isolate::InitWithoutSnapshot() 行 3308	C-
v8.dll!v8::Isolate::Initialize(v8::Isolate * isolate, const v8::Isolate::CreateParams & params) 行 8094	C-
v8.dll!v8::Isolate::New(const v8::Isolate::CreateParams & params) 行 8106	C-
d8.exe!v8::Shell::Main(int argc, char * * argv) 行 3514	C-
d8.exe!main(int argc, char * * argv) 行 3640	C-

Technical summary

(1) The number of Bytecode is the subscript of the Bytecode handler in the array dispatch_table_; **(2)**

The initialization of dispatch_table_ is completed when Isolate starts;

(3) The advantage of using fixed physical registers to save dispatch_table_ is to avoid unnecessary Pushing and popping simplifies the design of Bytecode and improves the efficiency

of Dispatch; Tip: When I debug V8, dispatch_table_ is always saved in the physical register R15. For debugging methods, see article 18.

Okay, that's it for today, see you next time.

Personal abilities are limited, there are shortcomings and

mistakes, criticisms and corrections are welcome WeChat: qq9123013 Note: v8 Communication Zhihu: https://www.zhihu.com/people/v8blink

This article was originally

published by Huidou and reprinted from: https://www.anquanke.com/post/id/

260182 Anquanke - Thoughtful new security media