

Solidity智能合约开发知识

第2.1课：数据类型基础

学习目标：掌握Solidity中的各种数据类型、理解值类型和引用类型的区别、学会使用运算符、掌握类型转换的安全方法

预计学习时间：2-2.5小时

难度等级：入门

目录

1. [数据类型概览](#)
2. [布尔类型](#)
3. [整数类型](#)
4. [地址类型](#)
5. [字节和字符串类型](#)
6. [枚举类型](#)
7. [运算符详解](#)
8. [类型转换](#)
9. [实践练习](#)

1. 数据类型概览

1.1 Solidity数据类型分类

在Solidity中，数据类型分为两大类：值类型（Value Types）和引用类型（Reference Types）。理解它们的区别对于编写高效、安全的智能合约至关重要。

值类型（Value Types）

值类型在赋值或传递时会创建一个完整的独立副本。修改副本不会影响原始值。

包含的类型：

- `bool`：布尔类型
- `int / uint`：整数类型
- `address`：地址类型
- `bytes1` 到 `bytes32`：固定大小字节数组
- `enum`：枚举类型

引用类型（Reference Types）

引用类型在赋值或传递时传递的是引用（内存地址），而不是完整的数据副本。修改引用会影响原始数据。

包含的类型：

- `array`：数组

- `string`：字符串
- `struct`：结构体
- `mapping`：映射
- `bytes`：动态字节数组

1.2 值类型与引用类型的对比

```
// 值类型示例
uint a = 10;
uint b = a; // 创建了a的副本
b = 20;      // 修改b不影响a
// 结果: a = 10, b = 20

// 引用类型示例
uint[] memory arr1 = new uint[](1);
arr1[0] = 10;
uint[] memory arr2 = arr1; // arr2指向arr1的同一块内存
arr2[0] = 20;             // 修改arr2会影响arr1
// 结果: arr1[0] = 20, arr2[0] = 20
```

值类型与引用类型的关键区别：

特性	值类型	引用类型
赋值方式	复制完整的值	传递引用（地址）
内存占用	每个变量独立占用内存	多个变量可能指向同一内存
修改影响	互不影响	修改一个会影响其他
Gas消耗	相对较低	相对较高
默认存储位置	无需指定	需要指定（storage/memory/calldata）

2. 布尔类型

2.1 布尔类型基础

布尔类型（`bool`）是最简单的数据类型，只有两个可能的值：`true`（真）和`false`（假）。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract BoolExample {
    bool public isActive = true;
    bool public isPaused = false;

    // 布尔类型的默认值是false
    bool public defaultBool; // 值为false
}
```

2.2 布尔运算符

Solidity支持以下布尔运算符：

逻辑运算符：

```
contract BoolOperators {
    function logicalOperators() public pure returns (bool, bool, bool, bool, bool) {
        bool a = true;
        bool b = false;

        return (
            !a,           // 逻辑非: false
            a && b,     // 逻辑与: false
            a || b,      // 逻辑或: true
            a == b,      // 等于: false
            a != b       // 不等于: true
        );
    }
}
```

运算符说明：

运算符	名称	示例	结果
!	逻辑非	!true	false
&&	逻辑与	true && false	false
	逻辑或	true false	true
==	等于	true == false	false
!=	不等于	true != false	true

2.3 布尔类型的实际应用

```
contract AccessControl {
    bool public isActive = true;
```

```

bool public isPaused = false;

// 检查系统状态
function checkActive() public view returns (bool) {
    return isActive && !isPaused;
}

// 切换状态
function toggleActive() public {
    isActive = !isActive;
}

// 条件判断
function executeIfActive() public view returns (string memory) {
    if (isActive && !isPaused) {
        return "System is active";
    } else {
        return "System is not active";
    }
}

```

3. 整数类型

3.1 整数类型概览

Solidity提供了两种整数类型：有符号整数（`int`）和无符号整数（`uint`）。

无符号整数（`uint`）：

无符号整数只能表示零和正数，不能表示负数。

```

uint8 // 0 到 255
uint16 // 0 到 65,535
uint32 // 0 到 4,294,967,295
uint64 // 0 到 18,446,744,073,709,551,615
uint128 // 0 到 2^128-1
uint256 // 0 到 2^256-1

// uint 等同于 uint256
uint public count; // 等同于 uint256 public count;

```

有符号整数（`int`）：

有符号整数可以表示负数、零和正数。

```

int8    // -128 到 127
int16   // -32,768 到 32,767
int32   // -2,147,483,648 到 2,147,483,647
int64   // -2^63 到 2^63-1
int128  // -2^127 到 2^127-1
int256  // -2^255 到 2^255-1

// int 等同于 int256
int public balance; // 等同于 int256 public balance;

```

3.2 为什么uint256最常用

很多初学者会疑惑：既然有uint8、uint16等更小的类型，为什么不用它们来节省空间？

答案：EVM的设计特性

以太坊虚拟机（EVM）是按照256位设计的，这意味着：

1. **EVM原生处理256位数据**: EVM内部的所有操作都是基于256位的
2. **使用较小类型需要额外操作**: 当使用uint8、uint16等类型时，EVM需要进行额外的截断和转换操作
3. **截断操作消耗更多gas**: 这些额外操作反而会增加gas消耗

实际测试对比：

```

contract GasComparison {
    uint256 public value256; // Gas: ~43,724
    uint128 public value128; // Gas: ~43,746 (更多! )
    uint8 public value8; // Gas: ~43,770 (最多! )
}

```

什么时候使用较小的整数类型？

只有在以下情况下才考虑使用较小的整数类型：

```

contract PackingExample {
    // 变量打包：多个小类型变量可以打包到同一个storage槽位
    uint128 public a; // 占用前128位
    uint128 public b; // 占用后128位
    // a和b共享同一个256位storage槽位，节省存储成本

    // 但如果单独使用，uint256更好
    uint256 public c; // 推荐
}

```

结论：

- 默认使用 `uint256`
- 需要负数时使用 `int256`
- 只有在变量打包优化时才考虑使用较小类型

3.3 整数运算

Solidity支持所有标准的算术运算：

```
contract IntegerOperations {
    function arithmeticOperations() public pure returns (uint, uint, uint, uint, uint, uint) {
        uint a = 10;
        uint b = 3;

        return (
            a + b,    // 加法: 13
            a - b,    // 减法: 7
            a * b,    // 乘法: 30
            a / b,    // 除法: 3 (注意: 只取整数部分)
            a % b,    // 取模: 1 (余数)
            a ** b    // 幂运算: 1000 (10的3次方)
        );
    }
}
```

重要提示：整数除法

Solidity没有浮点数类型，所有除法运算都是整数除法：

```
contract DivisionExample {
    function divide() public pure returns (uint) {
        uint result = 10 / 3;    // 结果是3, 不是3.333...
        return result;
    }

    // 如果需要精度, 需要使用定点数技巧
    function divideWithPrecision() public pure returns (uint) {
        uint numerator = 10 * 1000;    // 先乘以精度倍数
        uint denominator = 3;
        uint result = numerator / denominator;    // 3333
        // 实际值: 3.333 (需要在前端除以1000显示)
        return result;
    }
}
```

3.4 整数溢出保护

Solidity 0.8.0之前的问题：

在Solidity 0.8.0之前，整数运算可能发生溢出而不报错，这导致了很多安全漏洞。

```
// 0.8.0之前的危险代码
uint8 max = 255;
max = max + 1;    // 溢出到0 (循环)
```

Solidity 0.8.0+的自动保护：

从0.8.0版本开始，Solidity自动检查整数溢出：

```
contract OverflowProtection {
    function testOverflow() public pure returns (uint8) {
        uint8 max = 255;
        // 下面这行会导致交易回退
        return max + 1; // Error: Arithmetic operation underflowed or overflowed
    }

    function testUnderflow() public pure returns (uint8) {
        uint8 min = 0;
        // 下面这行会导致交易回退
        return min - 1; // Error: Arithmetic operation underflowed or overflowed
    }
}
```

unchecked关键字：

在某些特殊情况下，如果你确定不会溢出，可以使用 `unchecked` 来节省gas：

```
contract UncheckedExample {
    // 使用unchecked (谨慎使用!)
    function incrementUnchecked(uint x) public pure returns (uint) {
        unchecked {
            return x + 1; // 不检查溢出, 节省gas
        }
    }

    // 典型应用场景: 循环计数器
    function sumArray(uint[] memory arr) public pure returns (uint) {
        uint sum = 0;
        for (uint i = 0; i < arr.length; ) {
            sum += arr[i];
            unchecked {
                i++; // i不可能溢出, 使用unchecked节省gas
            }
        }
        return sum;
    }
}
```

何时使用 `unchecked`：

- 循环计数器（数组长度不可能达到uint256上限）
- 已经检查过不会溢出的计算
- 性能关键路径（需要节省gas）

警告：不正确使用 `unchecked` 可能导致严重的安全漏洞！

3.5 比较运算符

```

contract ComparisonOperators {
    function compare() public pure returns (bool, bool, bool, bool, bool, bool) {
        uint a = 10;
        uint b = 5;

        return (
            a == b, // 等于: false
            a != b, // 不等于: true
            a > b, // 大于: true
            a < b, // 小于: false
            a >= b, // 大于等于: true
            a <= b // 小于等于: false
        );
    }
}

```

4. 地址类型

4.1 地址类型基础

地址类型 (`address`) 是Solidity特有的类型，用于存储以太坊地址。一个地址是20字节（160位）的值。

两种地址类型：

```

contract AddressTypes {
    // 普通地址
    address public normalAddress;

    // 可支付地址
    address payable public payableAddress;
}

```

两种地址的区别：

特性	<code>address</code>	<code>address payable</code>
接收ETH	不可以	可以
transfer方法	没有	有
send方法	没有	有
余额查询	可以	可以
转换	不能转为payable	可以转为普通address

4.2 地址类型的常用功能

```

contract AddressFeatures {

```

```

// 查询地址余额
function getBalance(address addr) public view returns (uint) {
    return addr.balance; // 返回该地址的ETH余额 (单位: wei)
}

// 获取当前合约地址
function getContractAddress() public view returns (address) {
    return address(this);
}

// 获取合约余额
function getContractBalance() public view returns (uint) {
    return address(this).balance;
}

// 检查是否为零地址
function isZeroAddress(address addr) public pure returns (bool) {
    return addr == address(0);
    // address(0) = 0x000000000000000000000000000000000000000000000000000000000000000
}
}

```

4.3 特殊地址变量

Solidity提供了一些特殊的全局地址变量：

```

contract SpecialAddresses {
    function getSpecialAddresses() public view returns (address, address, address) {
        return (
            msg.sender,           // 当前调用者的地址
            tx.origin,           // 交易发起者的地址 (最原始的调用者)
            address(this)         // 当前合约的地址
        );
    }

    // msg.sender vs tx.origin的区别
    function demonstrateDifference() public view returns (string memory) {
        // 用户 -> 合约A -> 合约B
        // 在合约B中:
        // msg.sender = 合约A的地址
        // tx.origin = 用户的地址

        if (msg.sender == tx.origin) {
            return "Called directly by user";
        } else {
            return "Called by another contract";
        }
    }
}

```

重要安全提示：不要使用`tx.origin`进行权限验证，因为它容易受到钓鱼攻击！始终使用`msg.sender`。

4.4 转账功能

address payable 类型支持转账功能：

```
contract TransferExample {
    // 接收ETH的函数需要payable修饰符
    receive() external payable {}

    // transfer方法（推荐，失败会回退）
    function transferETH(address payable recipient, uint amount) public {
        recipient.transfer(amount); // 如果失败，整个交易回退
    }

    // send方法（不推荐，需要检查返回值）
    function sendETH(address payable recipient, uint amount) public returns (bool) {
        bool success = recipient.send(amount); // 失败返回false，不回退
        require(success, "Send failed");
        return success;
    }

    // call方法（最灵活，推荐用于转账）
    function callTransfer(address payable recipient, uint amount) public {
        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

三种转账方法的对比：

方法	Gas限制	失败处理	推荐程度
transfer	2300 gas	自动回退	中等
send	2300 gas	返回false	低
call	无限制	返回false	高（配合require）

4.5 地址类型转换

```
contract AddressConversion {
    // address转为address payable
    function toPayable(address addr) public pure returns (address payable) {
        return payable(addr);
    }

    // uint160转为address
    function uintToAddress(uint160 num) public pure returns (address) {
        return address(num);
    }
}
```

```

// address转为uint160
function addressToInt(address addr) public pure returns (uint160) {
    return uint160(addr);
}

// 示例：使用转换
function convertAndTransfer() public {
    address normalAddr = msg.sender;
    address payable payableAddr = payable(normalAddr);
    // 现在可以向payableAddr转账
}
}

```

为什么只能和uint160转换？

因为地址是20字节 = 160位，所以只能和uint160进行转换。

5. 字节和字符串类型

5.1 固定大小字节数组

固定大小字节数组有 bytes1 到 bytes32，共32种类型：

```

contract FixedBytes {
    bytes1 public b1 = 0x12;
    bytes4 public b4 = 0x12345678;
    bytes32 public b32 =
0x123456789012345678901234567890123456789012345678901234;

    // 获取长度（固定）
    function getLength() public pure returns (uint, uint, uint) {
        bytes1 a;
        bytes4 b;
        bytes32 c;
        return (a.length, b.length, c.length); // 1, 4, 32
    }

    // 访问单个字节
    function accessByte() public view returns (bytes1) {
        return b32[0]; // 访问第一个字节
    }
}

```

常见用途：

```

contract BytesUseCases {
    // 1. 存储哈希值
    bytes32 public fileHash;

    function storeHash(string memory data) public {

```

```

        fileHash = keccak256(bytes(data));
    }

    // 2. 存储签名
    bytes32 public r;
    bytes32 public s;
    uint8 public v;

    // 3. 紧凑数据存储
    bytes4 public functionSelector = 0x70a08231; // balanceOf(address)的函数选择器
}

```

`bytes32`最常用：

因为大多数哈希函数（如keccak256、SHA256）返回32字节的哈希值，所以`bytes32`是最常用的字节类型。

5.2 动态字节数组

`bytes`是动态长度的字节数组：

```

contract DynamicBytes {
    bytes public data;

    // 添加字节
    function pushByte() public {
        data.push(0x12);
    }

    // 获取长度
    function getLength() public view returns (uint) {
        return data.length;
    }

    // 访问元素
    function getByte(uint index) public view returns (bytes1) {
        require(index < data.length, "Index out of bounds");
        return data[index];
    }

    // 删除最后一个元素
    function popByte() public {
        data.pop();
    }
}

```

5.3 字符串类型

字符串 (`string`) 本质上是UTF-8编码的动态字节数组。

```

contract StringExample {
    string public name = "Solidity";
}

```

```

string public greeting;

// 设置字符串
function setGreeting(string memory _msg) public {
    greeting = _msg;
}

// 获取字符串
function getGreeting() public view returns (string memory) {
    return greeting;
}
}

```

字符串的限制：

Solidity的字符串类型功能有限：

```

contract StringLimitations {
    string public str1 = "Hello";
    string public str2 = "World";

    // 错误：不能直接比较
    // function compare() public view returns (bool) {
    //     return str1 == str2; // 编译错误!
    // }

    // 错误：不能直接获取长度
    // function getLength() public view returns (uint) {
    //     return str1.length; // 编译错误!
    // }

    // 错误：不能直接拼接（0.8.12之前）
    // function concat() public view returns (string memory) {
    //     return str1 + str2; // 编译错误!
    // }
}

```

5.4 字符串操作

字符串比较：

```

contract StringComparison {
    // 正确的字符串比较方法：比较哈希值
    function compareStrings(
        string memory a,
        string memory b
    ) public pure returns (bool) {
        return keccak256(bytes(a)) == keccak256(bytes(b));
    }

    // 使用示例
}

```

```

function testComparison() public pure returns (bool, bool) {
    return (
        compareStrings("Hello", "Hello"), // true
        compareStrings("Hello", "World") // false
    );
}
}

```

字符串拼接 (Solidity 0.8.12+) :

```

contract StringConcatenation {
    // 使用string.concat (0.8.12+)
    function concatenate(
        string memory a,
        string memory b
    ) public pure returns (string memory) {
        return string.concat(a, " ", b);
    }

    // 使用示例
    function testConcat() public pure returns (string memory) {
        return concatenate("Hello", "World"); // "Hello World"
    }

    // 拼接多个字符串
    function concatMultiple() public pure returns (string memory) {
        return string.concat("Hello", " ", "Solidity", " ", "World");
    }
}

```

字符串与bytes转换:

```

contract StringBytesConversion {
    // 字符串转bytes
    function stringToBytes(string memory str) public pure returns (bytes memory) {
        return bytes(str);
    }

    // 获取字符串长度 (通过转换为bytes)
    function getStringLength(string memory str) public pure returns (uint) {
        return bytes(str).length;
    }

    // bytes转字符串
    function bytesToString(bytes memory data) public pure returns (string memory) {
        return string(data);
    }
}

```

6. 枚举类型

6.1 枚举基础

枚举（enum）用于定义一组命名的常量，提高代码可读性。

```
contract EnumExample {
    // 定义枚举
    enum Status {
        Pending,      // 0
        Approved,     // 1
        Rejected,     // 2
        Cancelled     // 3
    }

    // 使用枚举
    Status public currentStatus;

    // 构造函数中设置初始状态
    constructor() {
        currentStatus = Status.Pending;
    }
}
```

枚举的特点：

1. 枚举值从0开始自动编号
2. 枚举本质上是 uint8 类型
3. 可以显式转换为整数
4. 提高代码可读性和类型安全

6.2 枚举操作

```
contract EnumOperations {
    enum OrderStatus {
        Created,      // 0
        Paid,         // 1
        Shipped,       // 2
        Delivered,     // 3
        Cancelled     // 4
    }

    OrderStatus public status;

    // 设置状态
    function createOrder() public {
        status = OrderStatus.Created;
    }

    function payOrder() public {
```

```

        require(status == OrderStatus.Created, "Order not created");
        status = OrderStatus.Paid;
    }

    function shipOrder() public {
        require(status == OrderStatus.Paid, "Order not paid");
        status = OrderStatus.Shipped;
    }

    // 检查状态
    function isPaid() public view returns (bool) {
        return status == OrderStatus.Paid;
    }

    // 枚举转整数
    function getStatusAsUint() public view returns (uint) {
        return uint(status);
    }

    // 整数转枚举（需要检查范围）
    function setStatusFromUint(uint _status) public {
        require(_status <= uint(OrderStatus.Cancelled), "Invalid status");
        status = OrderStatus(_status);
    }
}

```

6.3 枚举的实际应用

```

contract Crowdfunding {
    enum ProjectStatus {
        Fundraising, // 募资中
        Successful, // 成功
        Failed // 失败
    }

    ProjectStatus public status = ProjectStatus.Fundraising;
    uint public goal = 100 ether;
    uint public raised;

    function contribute() public payable {
        require(status == ProjectStatus.Fundraising, "Not fundraising");
        raised += msg.value;
    }

    function finalize() public {
        require(status == ProjectStatus.Fundraising, "Already finalized");

        if (raised >= goal) {
            status = ProjectStatus.Successful;
        } else {
            status = ProjectStatus.Failed;
        }
    }
}

```

```
    }
}
}
```

6.4 枚举的优势

1. 提高可读性：

```
// 使用枚举（清晰）
if (status == OrderStatus.Paid) {
    // ...
}

// 使用数字（不清晰）
if (status == 1) {
    // ...
}
```

2. 类型安全：

```
contract TypeSafety {
    enum Status { Pending, Approved, Rejected }
    Status public status;

    // 只能赋值为枚举中定义的值
    function setStatus() public {
        status = Status.Approved; // 正确
        // status = 10; // 编译错误
    }
}
```

3. 节省Gas：

枚举本质是 `uint8`，比使用 `string` 存储状态更省gas。

7. 运算符详解

7.1 算术运算符

```
contract ArithmeticOperators {
    function arithmetic() public pure returns (uint, uint, uint, uint, uint, uint) {
        uint a = 10;
        uint b = 3;

        return (
            a + b, // 加法: 13
            a - b, // 减法: 7
            a * b, // 乘法: 30
        );
    }
}
```

```

        a / b,    // 除法: 3
        a % b,    // 取模: 1
        a ** b    // 幂运算: 1000
    );
}

// 复合赋值运算符
function compoundAssignment() public pure returns (uint) {
    uint x = 10;
    x += 5;   // 等同于 x = x + 5
    x -= 3;   // 等同于 x = x - 3
    x *= 2;   // 等同于 x = x * 2
    x /= 4;   // 等同于 x = x / 4
    x %= 3;   // 等同于 x = x % 3
    return x;
}
}

```

7.2 比较运算符

```

contract ComparisonOperators {
    function comparison() public pure returns (bool, bool, bool, bool, bool, bool) {
        uint a = 10;
        uint b = 5;

        return (
            a == b,    // 等于: false
            a != b,   // 不等于: true
            a > b,    // 大于: true
            a < b,    // 小于: false
            a >= b,   // 大于等于: true
            a <= b    // 小于等于: false
        );
    }
}

```

7.3 逻辑运算符

```

contract LogicalOperators {
    function logical() public pure returns (bool, bool, bool) {
        bool a = true;
        bool b = false;

        return (
            a && b,   // 逻辑与: false
            a || b,   // 逻辑或: true
            !a        // 逻辑非: false
        );
    }
}

```

7.4 位运算符

位运算符直接操作整数的二进制位：

```
contract BitwiseOperators {
    function bitwise() public pure returns (uint, uint, uint, uint, uint, uint) {
        uint a = 5;      // 二进制: 0101
        uint b = 3;      // 二进制: 0011

        return (
            a & b,      // 按位与: 1 (0001)
            a | b,      // 按位或: 7 (0111)
            a ^ b,      // 按位异或: 6 (0110)
            ~a,         // 按位非: uint256最大值-5
            a << 1,     // 左移: 10 (1010)
            a >> 1     // 右移: 2 (0010)
        );
    }

    // 位运算的实际应用
    function checkBit(uint value, uint position) public pure returns (bool) {
        // 检查某一位是否为1
        return (value & (1 << position)) != 0;
    }

    function setBit(uint value, uint position) public pure returns (uint) {
        // 将某一位设置为1
        return value | (1 << position);
    }

    function clearBit(uint value, uint position) public pure returns (uint) {
        // 将某一位设置为0
        return value & ~(1 << position);
    }
}
```

7.5 短路运算

逻辑与 (`&&`) 和逻辑或 (`||`) 支持短路运算：

```
contract ShortCircuit {
    // 逻辑与的短路
    function andShortCircuit(uint x, uint y) public pure returns (bool) {
        // 如果x == 0, 不会执行y / x (避免除零错误)
        if (x != 0 && y / x > 5) {
            return true;
        }
        return false;
    }

    // 逻辑或的短路
}
```

```

function orShortCircuit(bool condition1, bool condition2) public pure returns (bool) {
    // 如果condition1是true, 不会检查condition2
    return condition1 || condition2;
}

// 短路运算的实际应用
function safeTransfer(address recipient, uint amount) public view returns (bool) {
    // 先检查简单条件, 再检查复杂条件 (优化gas)
    return recipient != address(0) && amount > 0 && address(this).balance >= amount;
}

```

短路运算的优势：

1. 防止错误：避免除零、数组越界等错误
2. 优化gas：避免不必要的计算
3. 逻辑清晰：先检查简单条件

8. 类型转换

8.1 隐式转换

隐式转换是编译器自动进行的转换，只在安全的情况下发生（小类型到大类型）：

```

contract ImplicitConversion {
    function implicitConvert() public pure returns (uint256, uint256) {
        uint8 small = 100;
        uint256 big = small; // 自动转换, 安全

        uint16 medium = 1000;
        uint256 big2 = medium; // 自动转换, 安全

        return (big, big2);
    }
}

```

隐式转换规则：

- 小整数类型可以隐式转换为大整数类型
- 无符号整数不能隐式转换为有符号整数
- 有符号整数不能隐式转换为无符号整数

8.2 显式转换

显式转换需要手动指定，用于大类型到小类型的转换：

```

contract ExplicitConversion {
    function explicitConvert() public pure returns (uint8) {
        uint256 big = 300;
        uint8 small = uint8(big); // 需要显式转换
        // 警告: 300转为uint8会溢出!
        // 结果: 44 (300 % 256 = 44)
        return small;
    }
}

```

危险示例:

```

contract DangerousConversion {
    function dangerousConvert() public pure returns (uint8, uint8) {
        uint256 value1 = 255;
        uint256 value2 = 256;

        return (
            uint8(value1), // 255 (正常)
            uint8(value2) // 0 (溢出!)
        );
    }
}

```

8.3 安全的类型转换

在进行显式转换时，应该先检查范围：

```

contract SafeConversion {
    // 安全转换函数
    function safeConvertToUint8(uint256 value) public pure returns (uint8) {
        require(value <= type(uint8).max, "Value too large for uint8");
        return uint8(value);
    }

    function safeConvertToInt16(uint256 value) public pure returns (uint16) {
        require(value <= type(uint16).max, "Value too large for uint16");
        return uint16(value);
    }

    // 使用type(T).max和type(T).min
    function getTypeInfo() public pure returns (uint8, uint8, int8, int8) {
        return (
            type(uint8).max, // 255
            type(uint8).min, // 0
            type(int8).max, // 127
            type(int8).min // -128
        );
    }
}

```

`type(T).max`和`type(T).min`:

```
contract TypeInfo {
    function showTypeInfo() public pure returns (
        uint8, uint8,
        uint16, uint16,
        uint256, uint256,
        int8, int8,
        int256, int256
    ) {
        return (
            type(uint8).min, // 0
            type(uint8).max, // 255
            type(uint16).min, // 0
            type(uint16).max, // 65535
            type(uint256).min, // 0
            type(uint256).max, // 2^256-1
            type(int8).min, // -128
            type(int8).max, // 127
            type(int256).min, // -2^255
            type(int256).max // 2^255-1
        );
    }
}
```

8.4 地址类型转换

```
contract AddressTypeConversion {
    // address转address payable
    function toPayable(address addr) public pure returns (address payable) {
        return payable(addr);
    }

    // uint160转address
    function uintToAddress(uint160 num) public pure returns (address) {
        return address(num);
    }

    // address转uint160
    function addressToInt(address addr) public pure returns (uint160) {
        return uint160(addr);
    }

    // 完整示例
    function conversionExample() public view returns (address, uint160, address payable) {
        address addr = msg.sender;
        uint160 num = uint160(addr);
        address payable pAddr = payable(addr);

        return (addr, num, pAddr);
    }
}
```

```
}
```

为什么是uint160?

因为地址是20字节 = 160位，所以只能和uint160互相转换。

9. 实践练习

练习1：投票合约

创建一个简单的投票合约，使用枚举定义投票选项。

任务要求：

1. 使用enum定义投票选项：Yes, No, Abstain
2. 使用mapping记录每个地址的投票
3. 使用uint统计每个选项的票数
4. 实现投票和查询功能

参考代码框架：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VotingSystem {
    // 1. 定义枚举
    enum Vote { Yes, No, Abstain }

    // 2. 状态变量
    mapping(address => Vote) public votes;
    mapping(address => bool) public hasVoted;
    uint public yesCount;
    uint public noCount;
    uint public abstainCount;

    // 3. 投票函数
    function vote(Vote _vote) public {
        // TODO: 实现投票逻辑
        // - 检查是否已投票
        // - 记录投票
        // - 更新计数
    }

    // 4. 查询函数
    function getResults() public view returns (uint, uint, uint) {
        return (yesCount, noCount, abstainCount);
    }

    function getMyVote() public view returns (Vote) {
        require(hasVoted[msg.sender], "You haven't voted");
        return votes[msg.sender];
    }
}
```

```
    }  
}
```

完整参考答案：

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
contract VotingSystem {  
    enum Vote { Yes, No, Abstain }  
  
    mapping(address => Vote) public votes;  
    mapping(address => bool) public hasVoted;  
    uint public yesCount;  
    uint public noCount;  
    uint public abstainCount;  
  
    event Voted(address indexed voter, Vote vote);  
  
    function vote(Vote _vote) public {  
        require(!hasVoted[msg.sender], "Already voted");  
  
        votes[msg.sender] = _vote;  
        hasVoted[msg.sender] = true;  
  
        if (_vote == Vote.Yes) {  
            yesCount++;  
        } else if (_vote == Vote.No) {  
            noCount++;  
        } else {  
            abstainCount++;  
        }  
  
        emit Voted(msg.sender, _vote);  
    }  
  
    function getResults() public view returns (uint, uint, uint) {  
        return (yesCount, noCount, abstainCount);  
    }  
  
    function getMyVote() public view returns (Vote) {  
        require(hasVoted[msg.sender], "You haven't voted");  
        return votes[msg.sender];  
    }  
  
    function getTotalVotes() public view returns (uint) {  
        return yesCount + noCount + abstainCount;  
    }  
}
```

练习2：类型转换练习

编写函数实现以下功能：

任务1：安全的uint256转uint8

```
function safeConvertToInt8(uint256 value) public pure returns (uint8) {
    // TODO: 添加范围检查
    // 如果value大于255，应该revert
}
```

任务2：字符串比较

```
function compareStrings(string memory a, string memory b)
    public pure returns (bool)
{
    // TODO: 实现字符串比较
    // 提示：使用keccak256
}
```

任务3：零地址检查

```
function isZeroAddress(address addr) public pure returns (bool) {
    // TODO: 检查是否为零地址
}
```

完整参考答案：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract TypeConversionPractice {
    // 任务1：安全转换
    function safeConvertToInt8(uint256 value) public pure returns (uint8) {
        require(value <= type(uint8).max, "Value too large for uint8");
        return uint8(value);
    }

    // 任务2：字符串比较
    function compareStrings(string memory a, string memory b)
        public pure returns (bool)
    {
        return keccak256(bytes(a)) == keccak256(bytes(b));
    }

    // 任务3：零地址检查
    function isZeroAddress(address addr) public pure returns (bool) {
        return addr == address(0);
    }

    // 额外测试函数
    function testConversion() public pure returns (uint8, uint8) {
```

练习3：综合练习 - 简单代币合约

创建一个简单的代币合约，综合运用所学的数据类型。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleToken {
    // 状态变量
    string public name = "My Token";
    string public symbol = "MTK";
    uint8 public decimals = 18;
    uint256 public totalSupply;

    address public owner;

    mapping(address => uint256) public balanceOf;

    // 事件
    event Transfer(address indexed from, address indexed to, uint256 value);

    // 构造函数
    constructor(uint256 _initialSupply) {
        owner = msg.sender;
        totalSupply = _initialSupply * 10 ** uint256(decimals);
        balanceOf[msg.sender] = totalSupply;
    }

    // 转账函数
}
```

```

function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0), "Cannot transfer to zero address");
    require(balanceOf[msg.sender] >= _value, "Insufficient balance");

    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;

    emit Transfer(msg.sender, _to, _value);
    return true;
}

// 查询余额
function getBalance(address _owner) public view returns (uint256) {
    return balanceOf[_owner];
}

// 铸造代币 (仅owner)
function mint(address _to, uint256 _amount) public {
    require(msg.sender == owner, "Only owner can mint");
    require(_to != address(0), "Cannot mint to zero address");

    totalSupply += _amount;
    balanceOf[_to] += _amount;

    emit Transfer(address(0), _to, _amount);
}
}

```

10. 常见问题解答

Q1：为什么字符串不能直接比较？

答：在Solidity中，字符串是引用类型，直接比较会比较引用（内存地址），而不是内容。正确的做法是比较它们的哈希值：

```
keccak256(bytes(str1)) == keccak256(bytes(str2))
```

Q2：uint和uint256有什么区别？

答：没有区别。`uint`是`uint256`的别名，它们完全等价。同样，`int`是`int256`的别名。

Q3：什么时候使用address，什么时候使用address payable？

答：

- 使用`address`：只需要存储地址或查询余额
- 使用`address payable`：需要向该地址转账ETH

可以使用`payable(addr)`将`address`转换为`address payable`。

Q4：为什么除法10/3结果是3而不是3.333？

答：Solidity没有浮点数类型，所有除法都是整数除法，只保留整数部分。如果需要精度，可以先乘以精度倍数再除：

```
uint result = (10 * 1000) / 3; // 3333
// 在前端显示时除以1000，得到3.333
```

Q5：unchecked什么时候使用？

答：只在以下情况使用unchecked：

1. 确定不会溢出的循环计数器
2. 已经通过其他方式检查过不会溢出的计算
3. 性能关键且安全性已验证的代码

不正确使用unchecked可能导致严重的安全漏洞！

Q6：bytes和string有什么区别？

答：

- bytes：原始字节数据，可以访问单个字节
- string：UTF-8编码的文本，不能访问单个字节

如果需要操作单个字节，使用bytes。如果存储文本，使用string。

Q7：枚举可以转换为整数吗？

答：可以。枚举本质上是uint8，可以显式转换：

```
enum Status { Pending, Approved }
Status s = Status.Approved;
uint num = uint(s); // 1
```

11. 知识点总结

数据类型分类

值类型：

- bool：布尔类型（true/false）
- int/uint：整数类型（有符号/无符号）
- address：地址类型（普通/可支付）
- bytes1-bytes32：固定字节数组
- enum：枚举类型

引用类型：

- array：数组

- string: 字符串
- struct: 结构体
- mapping: 映射
- bytes: 动态字节数组

关键特性

1. **uint256**最常用：EVM原生类型，最高效
2. **Solidity 0.8+**自动检查溢出：提高安全性
3. **address**是区块链特有类型：用于存储以太坊地址
4. 字符串比较需要用哈希：不能直接比较
5. 枚举提高可读性：类型安全且节省gas
6. 无浮点数：除法只保留整数部分

安全实践

1. 类型转换前检查范围
2. 使用 `type(T).max` 和 `type(T).min`
3. 注意整数除法特性
4. 谨慎使用 `unchecked`
5. 使用 `msg.sender` 而不是 `tx.origin`
6. 零地址检查

类型选择建议

整数类型：

- 默认使用 `uint256`
- 需要负数时使用 `int256`
- 只在变量打包时考虑小类型

地址类型：

- 普通地址用 `address`
- 需要接收ETH用 `address payable`

文本类型：

- 短标识符用 `bytes32`
- 用户输入文本用 `string`

状态管理：

- 有限状态集合用 `enum`

12. 学习检查清单

完成本课后，你应该能够：

数据类型理解：

- 区分值类型和引用类型
- 理解uint256为什么最常用
- 掌握地址类型的特性
- 理解枚举的优势

布尔和整数：

- 会使用布尔运算符
- 会使用整数运算符
- 理解整数溢出保护
- 知道何时使用unchecked

字符串和字节：

- 会比较字符串
- 会拼接字符串（0.8.12+）
- 理解bytes和string的区别
- 会使用bytes32存储哈希

枚举和转换：

- 会定义和使用枚举
- 会进行安全的类型转换
- 会使用type(T).max/min
- 理解地址类型转换

实践能力：

- 能编写投票合约
- 能实现类型转换函数
- 能综合运用各种类型

13. 下一步学习

完成本课后，建议：

1. 反复练习示例代码
2. 完成所有练习题
3. 尝试修改和扩展示例合约
4. 准备学习第2.2课：引用类型详解

下节课预告：

- 数组的详细用法
- 映射的特性和限制
- 结构体的定义和使用
- 存储位置的深入理解

14. 扩展资源

官方文档：

- Solidity数据类型：<https://docs.soliditylang.org/en/latest/types.html>
- Solidity运算符：<https://docs.soliditylang.org/en/latest/types.html#operators>

学习资源：

- Solidity by Example - Types：<https://solidity-by-example.org/primitives/>
- CryptoZombies Lesson 1-2：<https://cryptozombies.io>

工具推荐：

- Remix IDE：在线开发环境
- Etherscan：查看已部署合约
- OpenZeppelin：安全的合约库

进阶阅读：

- 以太坊黄皮书：深入理解EVM
- Solidity安全最佳实践
- Gas优化技巧

15. 完整示例合约

综合运用本课所学知识的完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/**
 * @title DataTypesDemo
 * @dev 演示Solidity各种数据类型的使用
 */
contract DataTypesDemo {

    // ===== 布尔类型 =====
    bool public isActive = true;

    // ===== 整数类型 =====
    uint256 public count = 100;
    int256 public balance = -50;
    uint8 public percentage = 75;

    // ===== 地址类型 =====
    address public owner;
    address payable public recipient;

    // ===== 字节类型 =====
    bytes32 public hash;
```

```
bytes public data;

// ===== 字符串类型 =====
string public name = "Solidity";
string public description;

// ===== 枚举类型 =====
enum Status { Pending, Approved, Rejected }
Status public currentStatus;

// ===== 事件 =====
event StatusChanged(Status newStatus);
event MessageUpdated(string newMessage);

// ===== 构造函数 =====
constructor() {
    owner = msg.sender;
    currentStatus = Status.Pending;
}

// ===== 布尔运算示例 =====
function checkActive() public view returns (bool) {
    return isActive && (count > 0);
}

function toggleActive() public {
    isActive = !isActive;
}

// ===== 整数运算示例 =====
function calculate(uint a, uint b) public pure returns (uint, uint, uint, uint) {
    return (
        a + b,    // 加法
        a * b,    // 乘法
        a / b,    // 除法
        a % b    // 取模
    );
}

function safeIncrement() public {
    // 使用checked算术 (默认)
    count = count + 1;
}

// ===== 地址操作示例 =====
function getBalance(address addr) public view returns (uint) {
    return addr.balance;
}

function isZero(address addr) public pure returns (bool) {
    return addr == address(0);
}
```

```
function setRecipient(address _recipient) public {
    require(_recipient != address(0), "Invalid address");
    recipient = payable(_recipient);
}

// ===== 字符串操作示例 =====
function setDescription(string memory _desc) public {
    description = _desc;
    emit MessageUpdated(_desc);
}

function compareStrings(string memory a, string memory b)
    public pure returns (bool)
{
    return keccak256(bytes(a)) == keccak256(bytes(b));
}

function concatenate(string memory a, string memory b)
    public pure returns (string memory)
{
    return string.concat(a, " ", b);
}

// ===== 字节操作示例 =====
function setHash(string memory input) public {
    hash = keccak256(bytes(input));
}

function addData(bytes1 b) public {
    data.push(b);
}

// ===== 枚举操作示例 =====
function approve() public {
    require(currentStatus == Status.Pending, "Not pending");
    currentStatus = Status.Approved;
    emit StatusChanged(Status.Approved);
}

function reject() public {
    require(currentStatus == Status.Pending, "Not pending");
    currentStatus = Status.Rejected;
    emit StatusChanged(Status.Rejected);
}

function getStatusAsUint() public view returns (uint) {
    return uint(currentStatus);
}

// ===== 类型转换示例 =====
function safeConvertToInt8(uint256 value)
```

```
public pure returns (uint8)
{
    require(value <= type(uint8).max, "Overflow");
    return uint8(value);
}

function addressToInt(address addr) public pure returns (uint160) {
    return uint160(addr);
}

// ===== 辅助函数 =====
function getInfo() public pure returns (uint8, uint256, int256) {
    return (
        type(uint8).max, // 255
        type(uint256).max, // 2^256-1
        type(int256).min // -2^255
    );
}
}
```

这个示例合约综合演示了：

- 所有基本数据类型的声明和使用
- 各种运算符的应用
- 类型转换的正确方法
- 字符串和字节的操作
- 枚举的实际应用
- 事件的使用

你可以在Remix中部署这个合约，测试所有函数，加深对数据类型的理解。