优化文档

由于学期末段大作业等诸多事项繁杂，笔者本身在优化方面投入的精力与取得的效果实在有限，所有的优化基本上是后端优化，许多中端优化并未进行，比如 Mem2Reg，死代码删除，GVN，公共子表达式删除等。最终笔者所进行的优化可以分为如下的三个部分，分别是指令选择，乘除法指令的优化，以及图着色寄存器分配。

1. 指令选择

这一步部分的所指，即选择addu subu addiu一类的指令或伪指令，避开 subiu 这一类实际开销大的伪指令，并且在处理运算型指令和判断型指令时，注意立即数的运算，当对应的运算数为立即数时，采用立即数类型指令，而不是让其占用一个临时寄存器之后再进行操作指令的构建。示例代码：

```
// li lw - binary
        if (num1 instanceof Constant &&
                (operator == Operator.ADD || operator == Operator.AND ||
                        operator == Operator.OR)) {
            reg2 = LiLw(num2, reg2);
            binaryImm(dstReg, reg2, ((Constant) num1).getNumber());
        } else if (num2 instanceof Constant &&
                (operator == Operator.ADD || operator == Operator.AND ||
                        operator == Operator.OR || operator ==
Operator.SUB)) {
            reg1 = LiLw(num1, reg1);
            binaryImm(dstReg, reg1, ((Constant) num2).getNumber());
        } else {
            reg1 = LiLw(num1, reg1);
            reg2 = LiLw(num2, reg2);
            binaryReg(dstReg, reg1, reg2);
        }
```

2. 乘除法优化

该部分的优化可以分成两个部分介绍，主要涉及的指令是 mult，div以及mod

乘法优化

乘法优化部分主要是对于特定的乘法式子进行优化，具体可以优化的情况如下：

- 乘数至少有一个为 0，这种情况的处理非常简单，即只需要将 dstReg设置为 0 即可。

```
MipsFactory.ins().buildLiLaInstr(LI, dstReg, 0);
```

- 乘数均为常数，这种情况，可以直接在编译器的代码中进行运算，直接将结果复制给对应寄存器。

```
int ans = ((Constant) val1).getNumber() * ((Constant) val2).getNumber();
MipsFactory.ins().buildLiLaInstr(LI, dstReg, ans);
```

- 乘数有一个为正负1，即

$$mult = \pm 1$$

同理，这种情况可以将另一个乘数的值通过 lw 或 li 或 move 指令赋值给目标寄存器。

```
if (MipsFactory.ins().getReg(val) == null) {
    oriLiLw(val, dstReg);
} else {
    MipsFactory.ins().buildMoveInstr(dstReg, MipsFactory.ins().getReg(val));
}
MipsFactory.ins().buildBinaryInstr(MipsBinaryInstr.Operator.SUBU,
        dstReg, Register.Reg.ZERO, dstReg);
```

- 乘数有一个为正负2的幂次， 即

$$mult = \pm 2^k$$

此种情况下的优化主要通过移位运算来实现，涉及的指令为 sll，即将另一个乘数左移 k 位即可完成对应的效果，再根据乘数的符号决定是否与 zero 寄存器相减。

```
reg1 = LiLw(val, reg1);
MipsFactory.ins().buildBiImInstr(MipsBiImInstr.Operator.SLL,
        dstReg, reg1, getBase(abs));
if (imm < 0) {
    MipsFactory.ins().buildBinaryInstr(MipsBinaryInstr.Operator.SUBU,
            dstReg, Register.Reg.ZERO, dstReg);
}
```

- 乘数有一个为正负2的幂次+-1或+-2，即

$$mult = \pm(2^k \pm 1) or \pm (2^k \pm 2)$$

这一部分的优化出发是基于原始乘法指令消耗较高的基础所设计的，我们不妨考虑一条乘法指令的代价为 4(mult) + 1(mflo)，因此考虑进符号可能得影响，我们对于

上述情况进行优化，达到降低指令代价的目的。主要的实现方式：**通过左移指令 sll 计算得到另一个乘数 x 乘以 2 的幂次的值，再和 x 自身进行对应的加减从而得到最终的结果。**示例代码如下：

```
reg1 = LiLw(val, reg1);
MipsFactory.ins().buildBiImInstr(MipsBiImInstr.Operator.SLL,
        T2, reg1, getBase(abs + 1));
MipsFactory.ins().buildBinaryInstr(MipsBinaryInstr.Operator.SUBU,
        dstReg, T2, reg1);
if (imm < 0) {
    MipsFactory.ins().buildBinaryInstr(MipsBinaryInstr.Operator.SUBU,
            dstReg, Register.Reg.ZERO, dstReg);
}
```

除法优化

除法优化的受益应该非常的高，毕竟一条 div 指令的代价高达 25，主要的实现方式是通过乘法 mult、移位 sra srl 来实现对于**除数为常数的除法指令**的优化，指令代价可以降到个位数。详细的数学原理可以参见文献：**division by invariant integers using multiplication**，文章中也给出了详细的伪代码图，实现过程可以分成两个部分（当然可以先对平凡的情况进行特判，比如常数除以常数）：

- 根据除数，所要求寄存器的精度位数选择好对应的乘数 `chooseMultipliers()`，伪代码图如下：

procedure CHOOSE_MULTIPLIER(**uword** d, **int** $prec$);
Cmt. d – Constant divisor to invert. $1 \le d < 2^N$.
Cmt. $prec$ – Number of bits of precision needed, $1 \le prec \le N$.
Cmt. Finds m, sh_{post}, ℓ such that:
Cmt. $\quad 2^{\ell-1} < d \le 2^{\ell}$.
Cmt. $\quad 0 \le sh_{post} \le \ell$. If $sh_{post} > 0$, then $N + sh_{post} \le \ell + prec$.
Cmt. $\quad 2^{N+sh_{post}} < m * d \le 2^{N+sh_{post}} * (1 + 2^{-prec})$.
Cmt. Corollary. If $d \le 2^{prec}$, then $m < 2^{N+sh_{post}} * (1 + 2^{1-\ell})/d \le 2^{N+sh_{post}-\ell+1}$.
Cmt. $\quad$ Hence m fits in $\max(prec, N-\ell)+1$ bits (unsigned).
Cmt.
int $\ell = \lceil \log_2 d \rceil$, $\quad sh_{post} = \ell$;
udword $m_{low} = \lfloor 2^{N+\ell}/d \rfloor$, $\quad m_{high} = \lfloor (2^{N+\ell} + 2^{N+\ell-prec})/d \rfloor$;
Cmt. To avoid numerator overflow, compute m_{low} as $2^N + (m_{low} - 2^N)$.
Cmt. Likewise for m_{high}. Compare m' in Figure 4.1.
Invariant. $m_{low} = \lfloor 2^{N+sh_{post}}/d \rfloor < m_{high} = \lfloor 2^{N+sh_{post}} * (1 + 2^{-prec})/d \rfloor$.
while $\lfloor m_{low}/2 \rfloor < \lfloor m_{high}/2 \rfloor$ **and** $sh_{post} > 0$ **do**
$\quad m_{low} = \lfloor m_{low}/2 \rfloor$; $\quad m_{high} = \lfloor m_{high}/2 \rfloor$; $\quad sh_{post} = sh_{post} - 1$;
end while; $\quad$ /* Reduce to lowest terms. */
return $(m_{high}, sh_{post}, \ell)$; $\quad$ /* Three outputs. */
end CHOOSE_MULTIPLIER;

Figure 6.2: Selection of multiplier and shift count

- 选择好对应的乘数之后，对于除数进行三种情况的判定，即是否为1， 是否为 2 的幂次，是否处于对应精度范围内

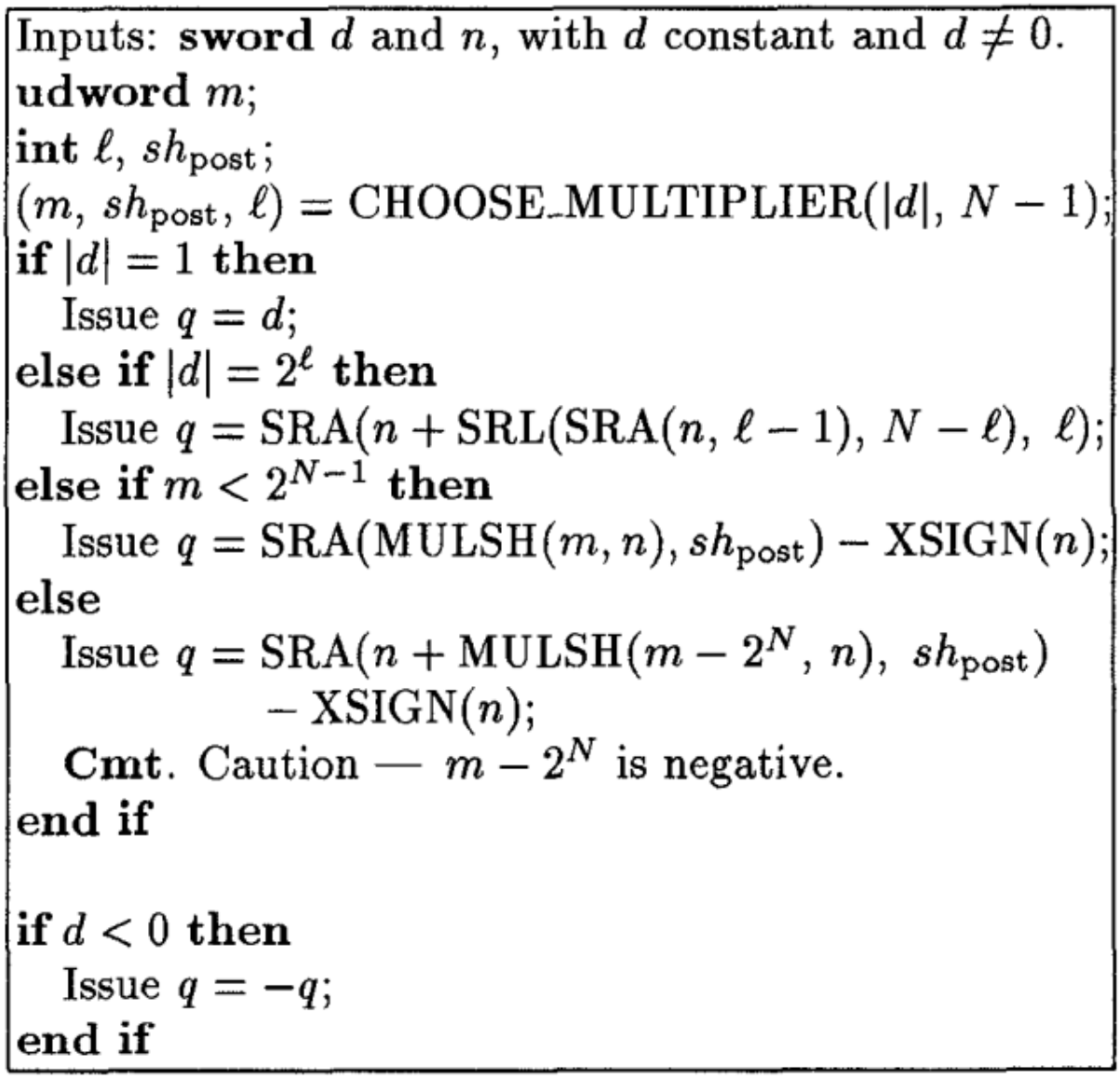


Figure 5.2: Optimized code generation of signed $q = \text{TRUNC}(n/d)$ for constant $d \neq 0$

取模优化

在朴素的实现中，取模运算实际上也是通过除法指令 div 来实现的，Mips 指令集通过 div 指令将余数储存在 HI 寄存器中，所以该指令的原始代价仍为 25(div) + 1(mfhi)。基于已经完成的乘除法优化，同样可以达到对应的优化效果，即先利用除法优化得到对应的商，然后被除数减去商与除数的积得到对应的余数，虽然整体计算过程看起来较为冗余，实际计算优化的空间，仍旧远小于 26 的原始指令代价。此处笔者因为涉及的不合理，没有使用乘法优化，下列代码仅供参考。

```
reg1 = divOpt(T2, reg1, reg2);
MipsFactory.ins().buildLiLaInstr(LI, T1, imm);
```

```
MipsFactory.ins().buildBinaryInstr(MipsBinaryInstr.Operator.MULT, T2, T1);
MipsFactory.ins().buildHiLoInstr(MipsHiLoInstr.Operator.MFLO, T1);
MipsFactory.ins().buildBinaryInstr(MipsBinaryInstr.Operator.SUBU, dstReg,
reg1, T1);
```

3. 图着色寄存器分配

坦率地来说，纵观所有的优化，合理的寄存器分配的优化效果肯定是属于头一档的，实际上从响应速度来说，寄存器的存取也是量级地快于内存的存取。笔者所采用的寄存器分配位于中端，主要的实现方式是将可以产生 def 的 llvm 指令在该 pass 中就分配好对应的全局寄存器，生成目标代码时直接将其与对应的寄存器挂钩，若没有分配到寄存器，再进行内存的存取。实现的思路为：**程序流图 Flow diagram 的划分 -> 活跃变量分析 Liveliness Analysis -> 图着色**
，主要的参考资料有：理论课上的ppt，**现代编译原理-c语言描述（虎书）**

程序流图 Flow diagram 的划分

LLVM 的语言规范已经要求我们在中端语言部分完成了基本块的划分，并保证了每个基本块的结束语句必须是一条跳转的语句，具体地来说就是 br 或 ret。此基础下。因此要实现程序流图的划分变得较为容易，也就是要建立每一个基本块与其前驱基本块的后继基本块的映射关系。实现方式为：**对于中端生成的指令序列进行一遍扫描，在每个基本块的结尾指令处进行流图生成，br 指令的目标基本块为当前基本块的后继基本块，同时也对称的设置目标基本块的前驱基本块。**至于跨函数的代码，实际上的函数跳转需要实现现场的保护，而寄存器的维护就是现场保护的重要一环，所以并不存在跨函数的前驱后继的基本块关系。代码示例：

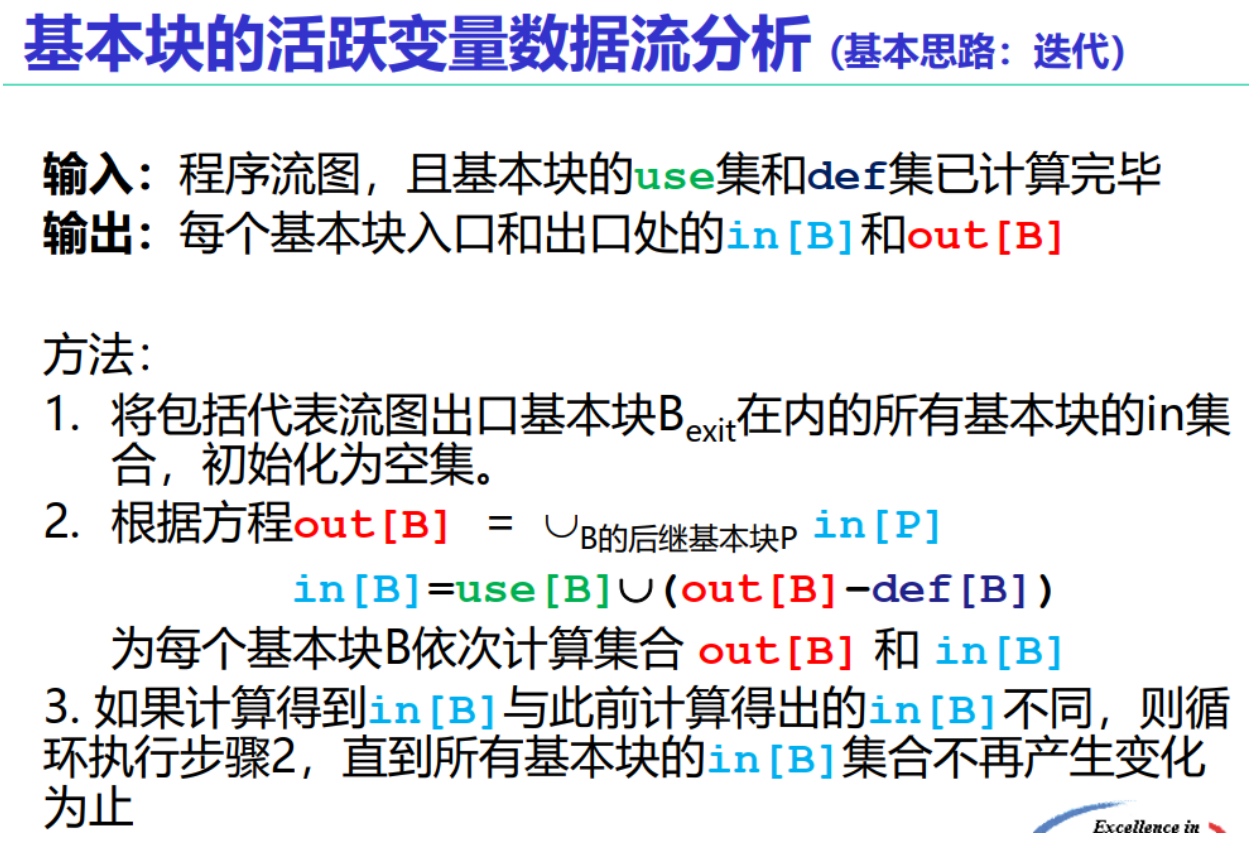
```
public void genFlow() {
        for (BasicBlock bb : basicBlocks) {
            ArrayList<Instr> instrs = bb.getInstructions();
            Instr last = instrs.get(instrs.size() - 1);
            if (last instanceof BrDirInstr) {
                BasicBlock tarBB = ((BrDirInstr) last).getTarBB();
                preMap.get(tarBB).add(bb);
                sucMap.get(bb).add(tarBB);
            } else if (last instanceof BrCondInstr) {
                BasicBlock ifBB = ((BrCondInstr) last).getIfBB();
                BasicBlock elseBB = ((BrCondInstr) last).getElseBB();
                sucMap.get(bb).add(ifBB);
                sucMap.get(bb).add(elseBB);
                preMap.get(elseBB).add(bb);
                preMap.get(ifBB).add(bb);
            }
```

```
        }
    }
```

活跃变量分析Liveliness Analysis

此处笔者的实现方式严格参照了理论课 PPT 上的基本实现流程，采用了**数据流分析**完成活跃变量分析。

- 生成基本块的 def-use集：首先需要生成每个基本块的 def-use 集合，代码的实现流程即对于基本块内的指令从头到尾进行扫描
  - 通过 Use 遍历当前指令所使用的所有 Value，如果该 Value 不属于 def 集且该 Value 不是 Function BasicBlock 一系列的 Value，则将其加入 use 集
  - 如果当前指令能够产生 def，即该指令需要分配寄存器，且该指令不属于 use 集，则将其加入 def 集
- 基于 def-use 集生成每个集的 in 集和 out 集，此处截取理论课 PPT 的代码思路：



无合并的图着色寄存器分配

该部分主要参考理论课 PPT 和虎书，实际的寄存器分配仍旧属于比较粗粒度的类型，主要的流程如下：

- 逐指令的**冲突图**绘制：每个基本块从后向前进行指令的遍历，如果该指令需要寄存器，即isDef，那么进行两个操作：
  - 该指令的 def 与当前 out 集合中的所有变量冲突，在冲突图上为其连上边
  - 将该指令的 def 从 out 集中删除，并将其所有的 usedValue 加入当前的 out 集

- 生成分配寄存器的节点队列与溢出节点队列：在此步对应指令是否进入对应队列的判定标准是**冲突中该点的度是否小于寄存器个数**，是则移入分配寄存器的节点队列 `NodesStack`，否则移入溢出节点队列 `spillList`（但后续分配寄存器时溢出节点队列的点仍有可能会分配到寄存器），具体的存储顺序是
  - `NodesStack`：点的度自栈底从小到大
  - `spillList`：点的度从头到尾从大到小

```
public void buildNodesStack(Function func) {
    HashMap<Value, HashSet<Value>> tmpAdj = new HashMap<>();
    tmpAdj.putAll(adjMatrix);
    spillList.clear();
    nodeStack.clear();

    while (!tmpAdj.isEmpty()) {
        ArrayList<Value> ascendingNode = sortVal(tmpAdj);
        stackOrSpill(ascendingNode, tmpAdj);
    }
}
```

- 图着色分配寄存器：
  - 首先需要将 `NodesStack` 中的所有点进行寄存器的分配，即每次 `pop()` 一个节点，得到其所有邻接顶点的寄存器颜色，选择与其不冲突的寄存器分配给当前节点。
  - 对于 `spillList` 中的点，尽管每个点的度数大于寄存器个数，但可能其邻接节点之间的寄存器相同，因此该寄存器仍可以分配空闲的寄存器。实际的流程即为：按照度从小到大的顺序遍历所有的节点，检验其邻接的所有节点是否占用了所有的寄存器，若否，则为其分配寄存器。

```
public void graphColoring(Function func) {
    var2Index.clear();
    processStackNodes();
    processSpilledNodes();
}
```

- 最后生成代码时，每进入一个新的 Function 即更新对应的指令-寄存器关系，并在生成后端代码时遵循对应的规则进行对应的分配。
  当然上述的生成过程非常的粗粒度，实际上的图着色可以非常的智能，涉及到合并

等多项操作，虎书中给出了如下的图，此处附上：

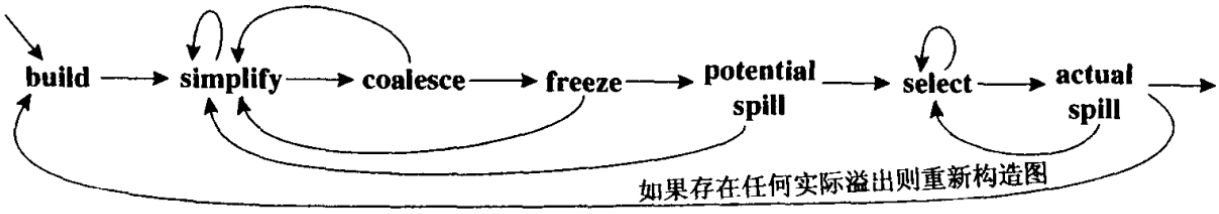


图 11-4　带合并的图着色

踩过的坑

在进行优化的过程中，特别是寄存器分配的过程中，笔者遇到的主要 bug 就是临时寄存器与全局寄存器的混用，**导致全局寄存器的值被修改**，此类 bug 可以导致各式各样的错误，毕竟谁也不知道被修改的寄存器中装的是地址还是操作数。在笔者的架构中，在 BinaryInstr 和 GEPInstr 的后端代码生成过程中，会涉及到频繁地使用临时寄存器，事先的设计过程中笔者也在生成后端代码的开头就声明使用的寄存器，诸如：

```
Register.Reg reg1 = Register.Reg.T1;
Register.Reg reg2 = Register.Reg.T2;
Register.Reg dstReg = (MipsFactory.ins().getReg(this) == null) ?
        Register.Reg.T0 : MipsFactory.ins().getReg(this);
```

在第一遍生成代码的过程中，笔者的代码直接将 `reg1` `reg2` 这类的寄存器当做了临时寄存器使用，但是当引入全局寄存器分配之后，`reg1` `reg2` 均可能被修改成全局寄存器，例如：

```
reg1 = LiLw(num1, reg1);
private Register.Reg LiLw(Value num, Register.Reg reg) {
    if (num instanceof Constant) {
        // Li
        MipsFactory.ins().buildLiLaInstr(LI, reg, ((Constant)
num).getNumber());
    } else if (MipsFactory.ins().getReg(num) != null) {
        return MipsFactory.ins().getReg(num);
    } else {
        // Lw
        StackInfo stackInfo =
MipsFactory.ins().getStackInfo(num.getMipsName());
        int offset = stackInfo.getOffset();
        MipsFactory.ins().buildMemInstr(MipsMem.Operator.LW, reg, offset,
Register.Reg.SP);
    }
    return reg;
}
```

此种情况下，如果在依赖于此前的代码生成逻辑将 `reg1` 当做临时寄存器使用，即将一些中间值存储在 `reg1` 中，就会导致**不可预想**的错误出现。因此，在寄存器的使用过程中，切记一定分清楚操作对象是临时寄存器还是全局寄存器，注意**维护全局寄存器的值**。