```
void myfunc(T&& tmprv)
{
    //T的类型不仅和调用者提供的实参(100)有关系。还和tmprv的类型(const T&)有关。

    //(2.2)万能引用：如果tmprv类型是个万能引用： T&&，能接受左值，也能接受右值，而且，你传递进来左值/右值，万能引用的表现是不
    int i = 18;       //int
    const int j = i;  //const int
    const int &k = i; //const int &
    myfunc(i);    //i是左值，T = int &   ,tmprv = int &
    myfunc(j);    //j是左值，T = const int &   ,tmprv =const  int &
    myfunc(k);    //k是个左值，T = const int &   ,tmprv =const  int &
    myfunc(100);  //100是个右值，T = int,  tmprv = int &&
```

```
// (2.3)传值方式：如果tmprv类型不是指针，也不是引用，而是常规的传值方式传递
//int i = 18;       //int
//const int j = i;  //const int
//const int &k = i; //const int &
//myfunc(i);    //T = int   ,tmprv = int
//myfunc(j);    //T = int   ,tmprv = int , const属性没传递进去，因为对方是新副本。
//myfunc(k);    //T = int   ,tmprv = int , const属性没传递进去，因为对方是新副本。
```

```
void myfunc(T tmprv)
{
    //T的类型不仅和调用者提供的实参(100)有关系。还和tmprv的类型(const T&)有关。

    //现在我们传递指针进去
    char mystr[] = "I Love China!";
    const char *const point = mystr; //第一个const表示point指向的目标中的内容不能通过point来改变。
                                     //第二个const表示point指向一个内容后，不可以再指向其他内容。
    myfunc(point);    //T=const char *,tmprv = const char *;
                      //第二个const没有了，第一个const保留。

    //总结：如果你传递的是const char *或者const char[]数组，那这个const会被保留。
```

```
//（2.4）数组做实参
//数组名字代表数组首地址。
```

```
//const char mystr[] = "I Love China!";
//myfunc(mystr); //T = const char [14], tmprv = const char (&)[14];
//T被推导成了数组类型,而tmprv中(&)代表该数组的一个引用

//（2.5）函数名做实参
//函数名相当于函数首地址，可以赋给一个函数指针
//myfunc(testFunc); // T = void(*)(void), tmprv = void(*)void  函数指针。
myfunc(testFunc);   //T = void(void),tmprv = void (&)void。   函数引用。
```

```
//总结：
//（1）推断中，引用类型实参的引用类型等于不存在；
//（2）万能引用，实参为左值或者右值，推断的结果不同；
//（3）按值传递的实参，传递给形参时const属性不起作用，则传递过去指针则另当别论。
//（4）数组或者函数类型再类型推断中被看做是指针，除非函数模板的形参是个引用。
```

```
//一：引用折叠 规则 ,以往掌握了 万能引用
int i = 18;       //i的类型是int型 ,i本身是个左值。
myfunc(100);      //i是右值,   T = int,  tmprv = int &&
myfunc(i);        //i是左值   ,T = int &,  tmprv = int &

void myfunc(int& &&tmprv) {..} //我们模拟编译器：认为这是编译器应该给咱们实例化出来的myfunc的样子。
                    //int&  是一组     , &&tmprv是第二组

void myfunc(int &tmprv) {...} //这个是编译器真正给我们实例化出来的myfunc模板函数的样子。

//引用折叠 ,c++11新标准 ，是一条规则（reference-collapsing rules），引用坍塌
//c++中有明确含义的引用只有两种，一种是&左值引用(类型)，一种是带&&右值引用(类型)。

//void myfunc(int& &&tmprv) //两组，第一组int&，左值引用   第二组 &&tmprv ,实际是右值引用个类型。
//分成两组的，  第一组：左值引用/右值引用      第二组 ：左值引用/右值引用
//左值引用  -- 左值引用   &     &
//左值引用  -- 右值引用   &     &&  (我们当前的情形)
//右值引用  -- 左值引用   &&    &
//右值引用  -- 右值引用   &&    &&
//折叠规则：如果任意一个引用为左值引用，那么结果就为左值引用(传染），否则就是右值引用。
```

```
//（1.1）引用的引用  : void func(int & &&i) {} //“i”：引用的引用非法
int b = 500;
int &byi = b; //byi是b引用
//int & &byy = byi; //非法  byy是byi的引用 ，引用的引用
//int & &byy = b;  //非法 ,引用的引用，有这个叫法，但是你不可以在程序中这么写，也就是  "& &"就不行。
//但是编译器内部在进行函数模板类型推断的时候，也可能出现 引用的引用 "& &"，这个时候编译器会用引用折叠规则处理（折叠规则消灭引用的引用的杀手）；
//但是编译器不允许你程序开发者直接写出引用的引用的这种形式的代码。

//需要折叠规则的这种场景有一些的：现在的函数模板实例化。以后我们研究auto，以后再具体讲解。
```