一、Block基础

1、Block简介

带有自动变量（局部变量）的匿名函数。它是C语言的扩充功能。之所以是拓展，是因为C语言不允许存在这样匿名函数。

关于“带有自动变量（局部变量）”的含义，这是因为Block拥有捕获外部变量的功能。在Block中访问一个外部的局部变量，Block会持用它的临时状态，自动捕获变量值，外部局部变量的变化不会影响它的的状态。

匿名函数是指不带函数名称的函数。

经典面试题如下：

```
int val = 10;
void (^blk)(void) = ^{
    printf("val=%d\n",val);
};
val = 2;
blk();
```

上面这段代码，输出值是：val = 10，而不是2。

block 在实现时会对它引用到的栈变量进行一次只读拷贝，然后在 block 块内使用该只读拷贝；换句话说block截获自动变量的瞬时值；或者block捕获的是自动变量的副本。

由于block捕获了自动变量的瞬时值，所以在执行block语法后，即使改写block中使用的自动变量的值也不会影响block执行时自动变量的值。

解决block不能修改自动变量值的办法是使用`__block`修饰符。

2、Block声明

① block的标准声明如下：

```
return_type (^blockName)(var_type) = ^return_type (var_type varName) {
    // ...
};
```

- return_type 表示返回的对象/关键字等(可以是void，并省略)
- blockName 表示block的名称
- var_type 表示参数的类型(可以是void，并省略)
- varName 表示参数名称

② typedef简化Block的声明如下：

```
typedef return_type (^BlockTypeName)(var_type);
```

二、Block用法

1、局部位置声明一个Block型的变量

```
return_type (^blockName)(var_type) = ^return_type (var_type varName) {
    // ...
};
blockName(var);
```

2、 @interface位置声明一个Block型的属性

```
@property(nonatomic, copy)return_type (^blockName) (var_type);
```

3、 在定义方法时，声明Block型的形参

```
- (void)yourMethod:(return_type (^)(var_type))blockName;
```

4、Block的内联用法

这种形式并不常用，匿名Block声明后立即被调用

```
^return_type (var_type varName)
{
    //...
}(var);
```

例子如下：

```
^void (NSString *string) {
        NSLog(@"String--%@",string);
 }(@"hello");
```

5、Block的递归调用

Block内部调用自身，递归调用是很多算法基础，特别是在无法提前预知循环终止条件的情况下。注意：由于Block内部引用了自身，这里必须使用__block避免循环引用问题。

```
static return_type (^blockName)(var_type) = ^return_type (var_type varName)
{
    if (returnCondition)
    {
        // ...
        // 【递归调用】
       blockName(varName);
    }
};
【初次调用】
blockName(varValue);
```

例子如下：

```
static void (^printHello)(NSInteger count) = ^void (NSInteger count) {
        if (count > 0) {
            NSLog(@"hello ---- %@",@(count));
            count--;
            printHello(count);
        }
    };
```

```
printHello(10);
```

6、Block作为返回值

方法的返回值是一个Block，可用于一些“工厂模式”的方法中：

```
- (return_type(^)(var_type))methodName
{
    return ^return_type(var_type param) {
        // ...
    };
}
```

例子：Masonry框架

三、Block应用场景

1、响应事件

情景：UIViewContoller有个UITableView并是它的代理，通过UITableView加载CellView。现在需要监听CellView中的某个按钮（可以通过tag值区分），并作出响应。

其实，即使Block不传递任何参数，也可以传递事件的。但这种情况，无法区分事件的激活方（cell里面的哪一个按钮？）。

2、传递数据

上面的响应事件，其实也是传递数据，只是它传递的对象是UIButton。传递数值或对象。

3、链式语法

链式编程思想：核心思想为将block作为方法的返回值，且返回值的类型为调用者本身，并将该方法以setter的形式返回，这样就可以实现了连续调用，即为链式编程。

四、Block循环引用

1、捕获变量机制

Block函数能捕获自动变量，但是在函数内部不能修改它，不然就是“编译错误”。Block定义时copy自动变量的值，在Block中作为常量使用，所以即使变量的值在Block外改变，也不影响他在Block中的值。想修改自动变量的值，需要使用__**Block**修饰符。

Block函数可以改变全局变量、静态变量、全局静态变量。因为全局变量或静态变量在内存中的地址是固定的，Block在读取改变量值的时候是直接从其所在的内存地址读取的，获取的是最新值，而不是在定义时copy的常量。

2、捕获对象机制

①全局变量和静态变量在内存中位置是固定的，Block在copy时不会retain对象。

②类实例变量在Block进行copy时不直接retain实例对象本身，但会retain类本身self，所以block可以直接读写类实例变量。

③自动变量在Block进行copy时会自动retain对象，增加引用计数。

④__block修饰对象在Block进行copy时不会retain。

3、循环引用及解决

Block可能会导致循环引用问题，因为block在拷贝到堆上的时候，会retain其引用的外部变量，那么如果block中如果引用了他的宿主对象，那很有可能引起循环引用。

MRC情况下，用__block可以消除循环引用。

ARC情况下，必须用弱引用才可以解决循环引用问题，iOS 5之后可以直接使用__weak，之前则只能使用__unsafe_unretained了，__unsafe_unretained缺点是指针释放后自己不会置空，造成野指针。

虽说使用`__weak`，但是此处会有一个隐患，你不知道 self 什么时候会被释放，为了保证在block内不会被释放，更多的时候需要添加`__strong`配合`strongSelf`使用。

有些情况下是可以直接使用self的，比如调用系统的方法。Block是否会造成循环引用要看是不是**相互持有强引用**。

4、变量知识补充

①全局变量

- 函数外面声明
- 可以跨文件访问
- 可以在声明时赋上初始值
- 如果没有赋初始值，系统自动赋值为0
- 存储位置：既非堆，也非栈，而是专门的【全局（静态）存储区static】！

②静态变量

- 函数外面 或 内部声明（即可修饰原全局变量亦可修饰原局部变量）
- 仅声明该变量的文件可以访问
- 可以在声明时赋上初始值
- 如果没有赋初始值，系统自动赋值为0
- 存储位置：既非堆，也非栈，而是专门的【全局（静态）存储区static】！

③局部变量（自动变量）

- 函数内部声明
- 仅当函数执行时存在
- 仅在本文件本函数内可访问
- 存储位置：自动保存在函数的每次执行的【栈帧】中，并随着函数结束后自动释放，另外，函数每次执行则保存在【栈】中

④内存分区

- 栈区（stack）：由编译器自动分配和释放，存放函数的参数值，局部变量的值等。操作方式类似于数据结构中的栈。
- 堆区（heap）：一般由程序员分配和释放，若程序员不释放，程序结束时可能由操作系统回收。与数据结构中的堆是两码事，分配方式类似于链表。
- 全局区（static）：全局变量和静态变量存放在此。
- 文字常量区：常量字符串放在此，程序结束后由系统释放。
- 程序代码区：存放函数体的二进制代码。

五、Block类型及内存管理

1、NSGlobalBlock：存储于程序数据区

Block 内部没有引用外部变量的 Block 类型都是 NSGlobalBlock 类型，存储于全局数据区，由系统管理其内存，retain、copy、release操作都无效。

注：针对没有捕获自动变量的block来说，虽然用clang的rewrite-objc转化后的代码中仍显示_NSConcretStackBlock，但是实际上不是这样的。

2、NSStackBlock：存储于栈区

这种情况，在非ARC下是无法编译的，在ARC下可以编译。

设置在栈上的Block，如果其作用域结束，该Block就被销毁。同样的，由于__block变量也配置在栈上，如果其作用域结束，则该__block变量也会被销毁。

3、NSMallocBlock：存储于堆区

堆中的Block无法直接创建，其需要由_NSConcreteStackBlock类型的Block**拷贝**而来。

4、Block在ARC与非ARC下的区别

- 在 ARC 中，捕获外部变量的 Block 的类型是 NSMallocBlock 或者 NSStackBlock，如果 Block 被赋值给了某个变量，在这个过程中会执行 _Block_copy 将原有的 NSStackBlock 变成 NSMallocBlock；但是如果 block 没有被赋值给某个变量，那它的类型就是 NSStackBlock；没有捕获外部变量的 Block 的类会是 NSGlobalBlock 即不在堆上，也不在栈上，它类似 C 语言函数一样会在代码段中。
- 在非 ARC 中，捕获了外部变量的 Block 的类会是 NSStackBlock，放置在栈上，没有捕获外部变量的 block 时与 ARC 环境下情况相同。
- 多数情况下，ARC下会默认把栈block被会直接拷贝生成到堆上。情况如下：

(1) 调用Block的copy实例方法时
(2) Block作为函数返回值返回时
(3) 将Block赋值给附有__strong修饰符id类型的类或Block类型成员变量时
(4) 将方法名中含有usingBlock的Cocoa框架方法或GCD的API中传递Block时

```objc
- (NSArray *)getBlockArray0 {
    int val = 10;
    return [NSArray arrayWithObjects:
            ^{NSLog(@"blk0:%d",val);},
            ^{NSLog(@"blk1:%d",val);}, nil];
}

- (void)testBlockForHeap0 {
    NSArray *tempArr = [self getBlockArray0];
    NSMutableArray *obj = [tempArr mutableCopy];
    typedef void (^blk_t) (void);
    blk_t block = (blk_t){[obj objectAtIndex:0]};
    block();
}
```

解决方案1：数组中的Block调用copy函数。

```objc
- (NSArray *)getBlockArray0 {
    int val = 10;
    return [NSArray arrayWithObjects:
            [^{NSLog(@"blk0:%d",val);} copy],
            [^{NSLog(@"blk1:%d",val);} copy], nil];
}

- (void)testBlockForHeap0 {
    NSArray *tempArr = [self getBlockArray0];
    NSMutableArray *obj = [tempArr mutableCopy];
    typedef void (^blk_t) (void);
    blk_t block = (blk_t){[obj objectAtIndex:0]};
    block();
}
```

解决方案2：数组中的Block直接添加到一个可变数组中

```objc
- (void)addBlockToArray:(NSMutableArray *)array {
    int val = 10;
    [array addObjectsFromArray:@[
                                 ^{NSLog(@"blk0:%d",val);},
                                  ^{NSLog(@"blk1:%d",val);}]];
}

- (void)testBlockForHeap2 {
    NSMutableArray *array = [NSMutableArray array];
    [self addBlockToArray:array];
    typedef void (^blk_t) (void);
    blk_t block = (blk_t){[array objectAtIndex:0]};
    block();
}
```

注意：在 ARC 开启的情况下，除非有上图中的例外，将只会有 NSConcreteGlobalBlock 和 NSConcreteMallocBlock 类型的 block。

五、Block底层实现

1、Block的作用是什么?

Block可以解决执行逻辑和上下文环境解耦的场景。

2、Block的本质是什么?

Block的本质是闭包功能在iOS上的实现。
Block本质上也是一个oc对象，他内部也有一个isa指针。
Block是封装了函数调用以及函数调用环境的OC对象。
Block是封装函数及其上下文的OC对象.
从设计模式的角度来考虑的话闭包就是一种策略模式(Strategy)的实现。

3、Block底层实现原理

```objc
#import <UIKit/UIKit.h>

int main(int argc, char * argv[]) {
    @autoreleasepool {
        int age = 10;
        void(^block)(int, int) = ^(int a, int b){
            NSLog(@"this is block, a = %d, b = %d",a,b);
            NSLog(@"this is block, ae = %d\n",age);
        };
        block(3, 5);
    }
}
```

main.m文件中加入如上所示代码
终端cd到文件main.m所在目录下，执行如下命令：
xcrun -sdk iphoneos clang -arch arm64 -rewrite-objc main.m

转化为C++实现后main函数内部结构如下：

```
int main(int argc, char * argv[]) {
    /* @autoreleasepool */ { __AtAutoreleasePool __autoreleasepool;
        int age = 10;
        //定义block变量代码
        void(*block)(int, int) = ((void (*)(int, int))&__main_block_impl_0((void
            *)__main_block_func_0, &__main_block_desc_0_DATA, age));
        //执行block内部的代码
        ((void (*)(__block_impl *, int, int))((__block_impl *)block)->FuncPtr)((__block_impl
            *)block, 3, 5);
    }
}
```

上述定义代码中，发现定义block变量代码调用了__main_block_impl_0函数，并且将__main_block_impl_0函数的地址赋值给了block。这里要了解几个关键定义如下：

① __main_block_impl_0

```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  int age;
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int _age, int flags=0) :
      age(_age) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
```

Block内代码块地址

存储着Block对象占用内存大小

__main_block_impl_0结构体内可以发现__main_block_impl_0构造函数中传入了四个参数。(void *)__main_block_func_0、&__main_block_desc_0_DATA、age、flags。其中flage有默认值，也就说flage参数在调用的时候可以省略不传。而最后的 age(_age)则表示传入的_age参数会自动赋值给age成员，相当于age = _age。 接下来着重看一下__main_block_impl_0函数内部三个参数分别代表什么。

② __block_impl

```
struct __block_impl {
  void *isa;
  int Flags;
  int Reserved;
  void *FuncPtr;
};
```

其结构体成员如下：

- isa，指向所属类的指针，也就是block的类型
- flags，标志变量，在实现block的内部操作时会用到

- Reserved，保留变量
- FuncPtr，block执行时调用的函数指针

可以看出，它包含了isa指针（包含isa指针的皆为对象），也就是说block也是一个对象(runtime里面，对象和类都是用结构体表示)。

③ __main_block_desc_0

```
static struct __main_block_desc_0 {
  size_t reserved;
  size_t Block_size;
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
```

其结构成员含义如下：

- reserved：保留字段
- Block_size：block大小(sizeof(struct __main_block_impl_0))

以上代码在定义__main_block_desc_0结构体时，同时创建了__main_block_desc_0_DATA，并给它赋值，以供在main函数中对__main_block_impl_0进行初始化。

④ (void *)__main_block_func_0

```
static void __main_block_func_0(struct __main_block_impl_0 *__cself, int a, int b) {
  int age = __cself->age; // bound by copy

            NSLog((NSString
                *)&__NSConstantStringImpl__var_folders_lc_9q1bn7vx3fb7kzch5kyjhqzc0000gn_T_main_490171_mi_0,a,b);
            NSLog((NSString
                *)&__NSConstantStringImpl__var_folders_lc_9q1bn7vx3fb7kzch5kyjhqzc0000gn_T_main_490171_mi_1,age);
        }
```

在__main_block_func_0函数中首先取出block中age的值，紧接着可以看到两个熟悉的NSLog，可以发现这两段代码恰恰是我们在block块中写下的代码。也就是说写在block块中的代码被封装成__main_block_func_0函数，并将__main_block_func_0函数的地址(void *)__main_block_func_0传入了__main_block_impl_0的构造函数中保存在__block_impl结构体内FuncPtr指针中。

⑤ age

age是定义的局部变量。因为在block块中使用到age局部变量，所以在block定义的时候会将age作为参数传入，也就是说block会捕获age，如果没有在block中使用age，这里将只会传入(void *)__main_block_func_0，&__main_block_desc_0_DATA两个参数。

由于block在定义的时候已经捕获到age并存储在__main_block_imp_0结构体中，调用block的时候会从block结构体中进行读取，因此在block定义之后对局部变量进行修改不影响block代码块中读取的值。

了解了block变量的定义后，来看执行block内部的代码。通过C++实现的main.m代码中发现调用block是通过block找到FunPtr直接调用，通过上面分析我们知道block指向的是__main_block_impl_0类型结构体，但是__main_block_impl_0结构体中并不能直接找到FunPtr，而FunPtr是存储在__block_impl中的，为什么block可以直接调用__block_impl中的FunPtr呢?

代码中可以看到(__block_impl *)block将block强制转化为__block_impl类型的，因为__block_impl是__main_block_impl_0结构体的第一个成员，相当于将__block_impl结构体的成员直

接拿出来放在__main_block_impl_0中，那么也就说明__block_impl的内存地址就是__main_block_impl_0结构体的内存地址开头。所以可以转化成功。并找到FunPtr成员。

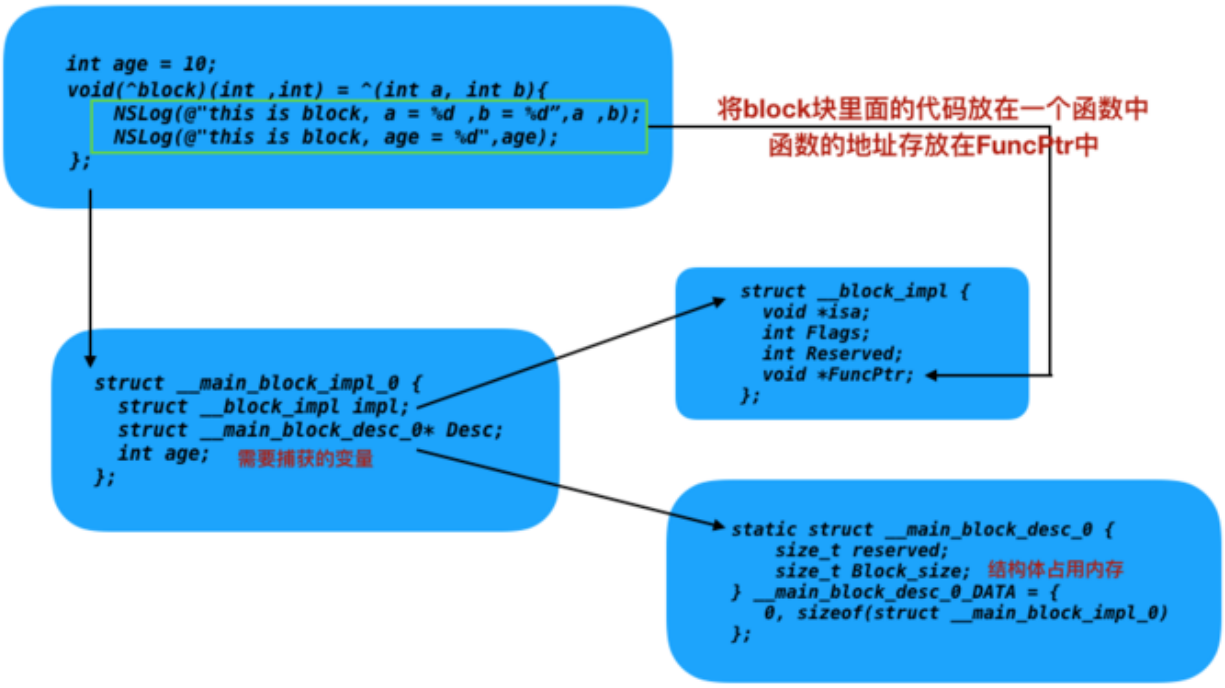
FunPtr中存储着通过代码块封装的函数地址，那么调用此函数，也就是会执行代码块中的代码。并且回头查看__main_block_func_0函数，可以发现第一个参数就是__main_block_impl_0类型的指针。也就是说将block传入__main_block_func_0函数中，便于重中取出block捕获的值。

通过上面对__main_block_impl_0结构体构造函数三个参数的分析我们可以得出结论：

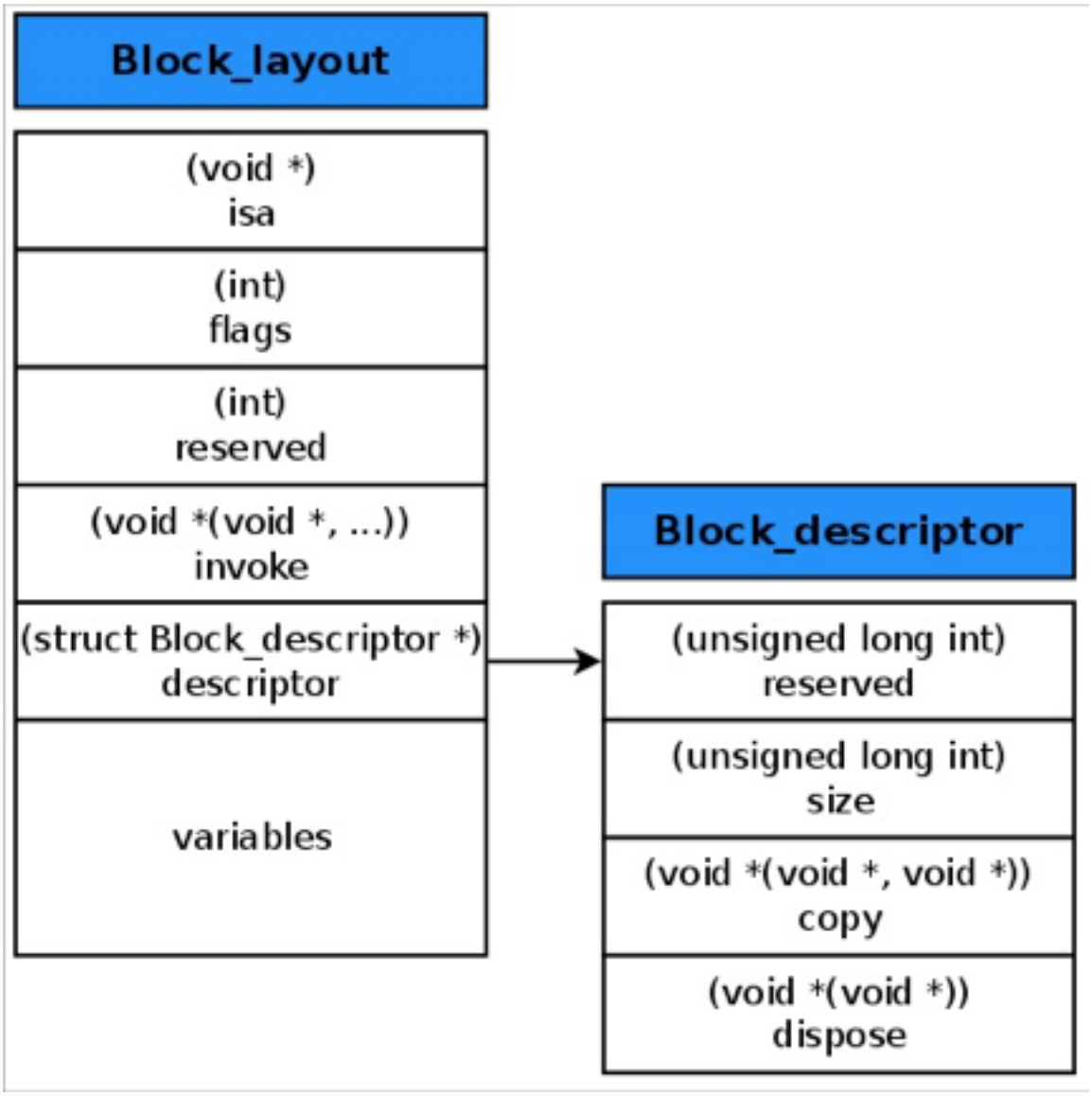
1. __block_impl结构体中isa指针存储着&_NSConcreteStackBlock地址，可以暂时理解为其类对象地址，block就是_NSConcreteStackBlock类型的。
2. block代码块中的代码被封装成__main_block_func_0函数，FuncPtr则存储着__main_block_func_0函数的地址。
3. Desc指向__main_block_desc_0结构体对象，其中存储__main_block_impl_0结构体所占用的内存。

总结

此时已经基本对block的底层结构有了基本的认识，上述代码可以通过一张图展示其中各个结构体之间的关系。



block底层的数据结构也可以通过一张图来展示



4、Block变量捕获机制

① Block捕获基本数据类型，修改main.m代码如下：

```objc
#import <stdio.h>
//全局变量
int a = 10;
//全局静态变量
static int b = 20;
int main(int argc, char * argv[]) {
    @autoreleasepool {
        //自动变量
        auto int c = 30;
        //静态变量
        static int d = 40;
        void(^block)(void) = ^{
            printf("hello, a = %d, b = %d, c = %d, d = %d",a,b,c,d);
        };
        a = 11;
        b = 21;
        c = 31;
        d = 41;
        block();
    }
}
```

转换为C++实现如下：

```objc
int a = 10;

static int b = 20;

struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  int c;c为值
  int *d;d为指针
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int _c, int *_d, int flags=0) : c(_c), d(_d) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
  int c = __cself->c; // bound by copy
  int *d = __cself->d; // bound by copy

            printf("hello, a = %d, b = %d, c = %d, d = %d",a,b,c,(*d));函数中直接调用a、b，d传入的是指针
        }

static struct __main_block_desc_0 {
  size_t reserved;
  size_t Block_size;
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
int main(int argc, char * argv[]) {
    /* @autoreleasepool */ { __AtAutoreleasePool __autoreleasepool;

        auto int c = 30;

        static int d = 40;
        void(*block)(void) = ((void (*)())&__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA,
            c, &d));
        a = 11;
        b = 21;
        c = 31;
        d = 41;
        ((void (*)(__block_impl *))((__block_impl *)block)->FuncPtr)((__block_impl *)block);
    }
}
```

结构体中没有a、b

传入的是d的地址

从上述C++实现代码看出：

- a、b两个变量没有被添加到结构体中，得出block不需要捕获全局变量，因为全局变量无论在哪里都可以访问。
- c、d两个变量都被捕获到block内部，但是c传入的是值，而d传入的则是地址。造成这种差异是因为自动变量可能会销毁，block在执行的时候有可能自动变量已经被销毁了，那么此时如果再去访问被销毁的地址肯定会发生坏内存访问，因此对于自动变量一定是值传递而不可能是指针传递了。而静态变量不会被销毁，所以完全可以传递地址。而因为传递的是值得地址，所以在block调用之前修改地址中保存的值，block中的地址是不会变得。所以值会随之改变。

总结如下：

变量类型		捕获到block内部	访问方式
局部变量	auto	√	值传递
	static	√	指针传递
全局变量		×	直接访问

局部变量都会被block捕获，自动变量是值捕获，静态变量为地址捕获，全局变量则不会被block捕获。

② Block捕获对象类型，修改main.m代码如下：

```
@interface Person : NSObject

@property (nonatomic, copy) NSString *name;

@end

@implementation Person

- (void)test {
    void(^block)(void) = ^ {
        NSLog(@"%@",self);
        NSLog(@"%@",self.name);
        NSLog(@"%@",self->_name);
    };
    block();
}

+ (void) test2 {
    NSLog(@"类方法test2");
}

- (instancetype)initWithName:(NSString *)name {
    if (self = [super init]) {
        self.name = name;
    }
    return self;
}

@end

int main(int argc, char * argv[]) {
    @autoreleasepool {
        Person *person = [[Person alloc] initWithName:@"block"];
        [person test];
        [Person test2];
//        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate
    class]));
    }
}
```

转换为C++实现如下：

```
struct __Person__test_block_impl_0 {
  struct __block_impl impl;
  struct __Person__test_block_desc_0* Desc;
  Person *self;
  __Person__test_block_impl_0(void *fp, struct __Person__test_block_desc_0 *desc, Person *_self, int flags=0) :
      self(_self) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
static void __Person__test_block_func_0(struct __Person__test_block_impl_0 *__cself) {
  Person *self = __cself->self; // bound by copy

        NSLog((NSString
            *)&__NSConstantStringImpl__var_folders_lc_9q1bn7vx3fb7kzch5kyjhqzc0000gn_T_main_78d1db_mi_0,self);
        NSLog((NSString *)&__NSConstantStringImpl__var_folders_lc_9q1bn7vx3fb7kzch5kyjhqzc0000gn_T_main_78d1db_mi_1,
            ((NSString *(*)(id, SEL))(void *)objc_msgSend)((id)self, sel_registerName("name")));
        NSLog((NSString *)&__NSConstantStringImpl__var_folders_lc_9q1bn7vx3fb7kzch5kyjhqzc0000gn_T_main_78d1db_mi_2,
            (*(NSString **)((char *)self + OBJC_IVAR_$_Person$_name)));
    }
static void __Person__test_block_copy_0(struct __Person__test_block_impl_0*dst, struct
    __Person__test_block_impl_0*src) {_Block_object_assign((void*)&dst->self, (void*)src->self, 3/
    *BLOCK_FIELD_IS_OBJECT*/);}

static void __Person__test_block_dispose_0(struct __Person__test_block_impl_0*src) {_Block_object_dispose((void*)src-
    >self, 3/*BLOCK_FIELD_IS_OBJECT*/);}

static struct __Person__test_block_desc_0 {
  size_t reserved;
  size_t Block_size;
  void (*copy)(struct __Person__test_block_impl_0*, struct __Person__test_block_impl_0*);
  void (*dispose)(struct __Person__test_block_impl_0*);
} __Person__test_block_desc_0_DATA = { 0, sizeof(struct __Person__test_block_impl_0), __Person__test_block_copy_0,
    __Person__test_block_dispose_0};

static void _I_Person_test(Person * self, SEL _cmd) {
    void(*block)(void) = ((void (*)())&__Person__test_block_impl_0((void *)__Person__test_block_func_0,
        &__Person__test_block_desc_0_DATA, self, 570425344));
    ((void (*)(__block_impl *))((__block_impl *)block)->FuncPtr)((__block_impl *)block);
}


static void _C_Person_test2(Class self, SEL _cmd) {
    NSLog((NSString *)&__NSConstantStringImpl__var_folders_lc_9q1bn7vx3fb7kzch5kyjhqzc0000gn_T_main_78d1db_mi_3);
}
```


从上图中可以发现：

- block只捕获了self变量
- 实例对象通过不同的方式去获取使用到的属性
- 不论对象方法还是类方法都默认传递了两个参数self和_cmd，既然self是作为参数传入，那么self肯定是局部变量。

5、Block类型

修改main.m代码如下

```objc
int main(int argc, char * argv[]) {
    @autoreleasepool {
        // __NSGlobalBlock__ : __NSGlobalBlock : NSBlock : NSObject
        void (^block)(void) = ^{
            NSLog(@"Hello");
        };
        NSLog(@"%@", [block class]);
        NSLog(@"%@", [[block class] superclass]);
        NSLog(@"%@", [[[block class] superclass] superclass]);
        NSLog(@"%@", [[[[block class] superclass] superclass] superclass]);
        NSLog(@"%@", [[[[[block class] superclass] superclass] superclass]
            superclass]);
//        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate
    class]));
    }
}
```

```
2019-01-18 14:10:28.174215+0800 BlockDemo[63744:1389138] __NSGlobalBlock__
2019-01-18 14:10:28.175169+0800 BlockDemo[63744:1389138] __NSGlobalBlock
2019-01-18 14:10:28.175276+0800 BlockDemo[63744:1389138] NSBlock
2019-01-18 14:10:28.175415+0800 BlockDemo[63744:1389138] NSObject
2019-01-18 14:10:28.175494+0800 BlockDemo[63744:1389138] (null)
```

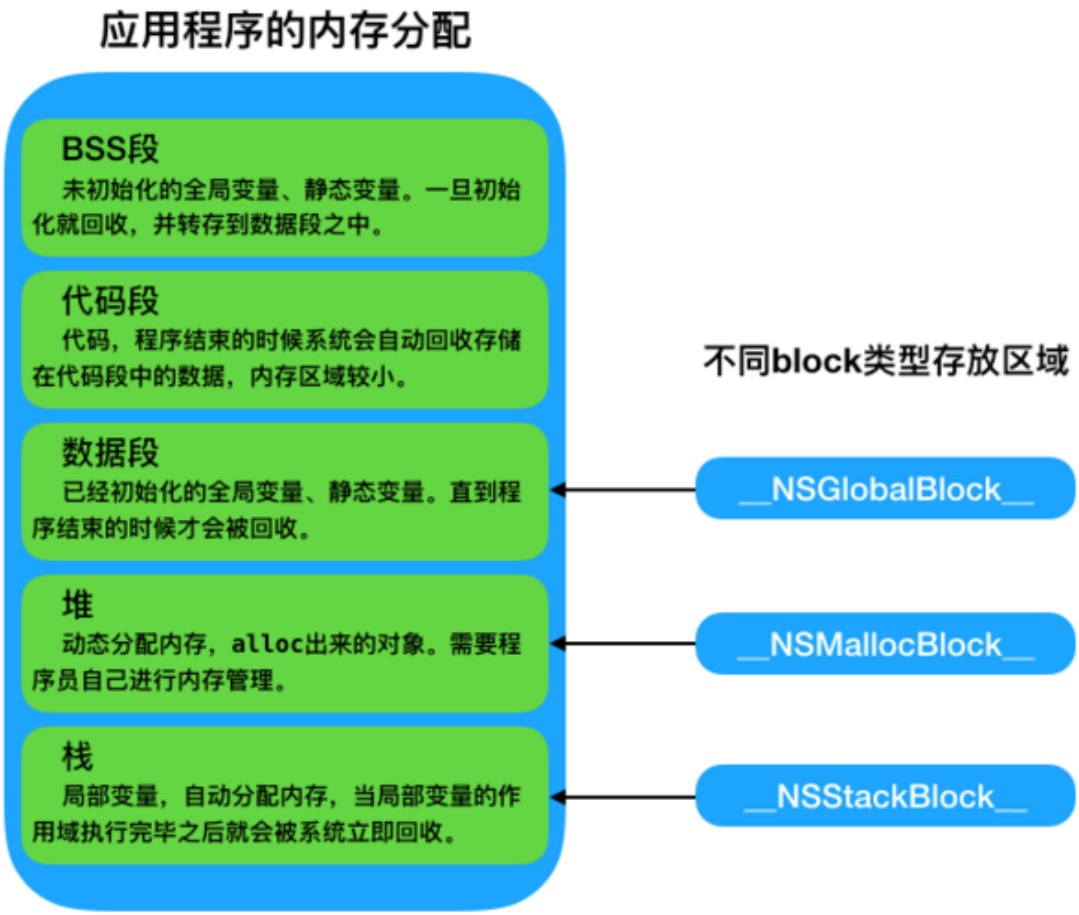
```objc
int main(int argc, char * argv[]) {
    @autoreleasepool {
        // 1. 内部没有调用外部变量的block
        void (^block1)(void) = ^{
            NSLog(@"Hello");
        };
        // 2. 内部调用外部变量的block
        int a = 10;
        void (^block2)(void) = ^{
            NSLog(@"Hello - %d",a);
        };
        // 3. 直接调用的block的class
        NSLog(@"%@ %@ %@", [block1 class], [block2 class], [^{
            NSLog(@"%d",a);
        } class]);
//        return UIApplicationMain(argc, argv, nil, NSStringFromCla
    class]));
    }
}
```

```
2019-01-18 15:39:04.297252+0800 BlockDemo[65246:1427119] __NSGlobalBlock__ __NSMallocBlock__ __NSStackBlock__
```

第一个图中可以看出block最终都是继承自NSBlock类型，而NSBlock继承于NSObjcet。那么block其中的isa指针其实是来自NSObject中的。这也更加印证了block的本质其实就是OC对象。

第二个图中代码转化为c++代码查看源码时却发现block的类型与打印出来的类型不一样，c++源码中三个block的isa指针全部都指向_NSConcreteStackBlock类型地址。可以猜测runtime运行时过程中也许对类型进行了转变。最终类型当然以runtime运行时类型也就是我们打印出的类型为准。

通过下面一张图看一下不同block的存放区域



上图中可以发现，根据block的类型不同，block存放在不同的区域中。

数据段中的__NSGlobalBlock__直到程序结束才会被回收，不过我们很少使用到__NSGlobalBlock__类型的block，因为这样使用block并没有什么意义。

__NSStackBlock__类型的block存放在栈中，我们知道栈中的内存由系统自动分配和释放，作用域执行完毕之后就会被立即释放，而在相同的作用域中定义block并且调用block似乎也多此一举。

__NSMallocBlock__是在平时编码过程中最常使用到的。存放在堆中需要我们自己进行内存管理。

注1：C++中void *作用

C++是强类型的语言。void的字面意思是“无类型”，void *则为“无类型指针”，void *可以指向任何类型的数据。 即任何类型的指针都可以直接赋值给void *，无需进行强制类型转换。但是void *必须要进行强制类型转换地赋给其它类型的指针。

注2：C++中&操作符的作用

①&(引用)：出现在变量声明语句中位于变量左边时,表示声明的是引用.

例如: int &rf; // 声明一个int型的引用rf。

②&(取地址运算符)：在给变量赋初值时出现在等号右边或在执行语句中作为一元运算符出现时表示取对象的地址。例如：

```
int a=3；
int &b=a；          //引用
int *p=&a;        //取地址
```