上一篇文章 iOS底层原理总结 - 探寻block的本质1 中已经介绍过block的底层本质实现以及了解了变量的捕获，本文继续探寻block的本质。

block对对象变量的捕获

block一般使用过程中都是对对象变量的捕获，那么对象变量的捕获同基本数据类型变量相同吗?

查看一下代码思考：当在block中访问的为对象类型时，对象什么时候会销毁?

```
typedef void (^Block)(void);
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Block block;
        {
            Person *person = [[Person alloc] init];
            person.age = 10;

            block = ^{
                NSLog(@"------block内部%d",person.age);
            };
        } // 执行完毕，person没有被释放
        NSLog(@"--------");
    } // person 释放
    return 0;
}
```

大括号执行完毕之后，`person`依然不会被释放。上一篇文章提到过，`person`为`auto`变量，传入的`block`的变量同样为`person`，即`block`有一个强引用引用`person`，所以`block`不被销毁的话，`peroson`也不会销毁。 查看源代码确实如此

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

将上述代码转移到MRC环境下，在MRC环境下即使block还在，`person`却被释放掉

了。因为MRC环境下block在栈空间，栈空间对外面的`person`不会进行强引用。

```
//MRC环境下代码
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Block block;
        {
            Person *person = [[Person alloc] init];
            person.age = 10;
            block = ^{
                NSLog(@"------block内部%d",person.age);
            };
            [person release];
        } // person被释放
        NSLog(@"--------");
    }
    return 0;
}
```

block调用copy操作之后，person不会被释放。

```
block = [^{
   NSLog(@"------block内部%d",person.age);
} copy];
```

上文中也提到过，只需要对栈空间的`block`进行一次`copy`操作，将栈空间的`block`拷贝到堆中，`person`就不会被释放，说明堆空间的`block`可能会对`person`进行一次`retain`操作，以保证`person`不会被销毁。堆空间的`block`自己销毁之后也会对持有的对象进行`release`操作。

也就是说栈空间上的block不会对对象强引用，堆空间的block有能力持有外部调用的对象，即对对象进行强引用或去除强引用的操作。

__weak

__weak添加之后，`person`在作用域执行完毕之后就被销毁了。

```
typedef void (^Block)(void);
```

```objc
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Block block;
        {
            Person *person = [[Person alloc] init];
            person.age = 10;

            __weak Person *waekPerson = person;
            block = ^{
                NSLog(@"------block内部%d",waekPerson.age);
            };
        }
        NSLog(@"--------");
    }
    return 0;
}
```

将代码转化为c++来看一下上述代码之间的差别。__weak修饰变量，需要告知编译器使用RAC环境及版本号否则会报错，添加说明`-fobjc-arc -fobjc-runtime=ios-8.0.0`

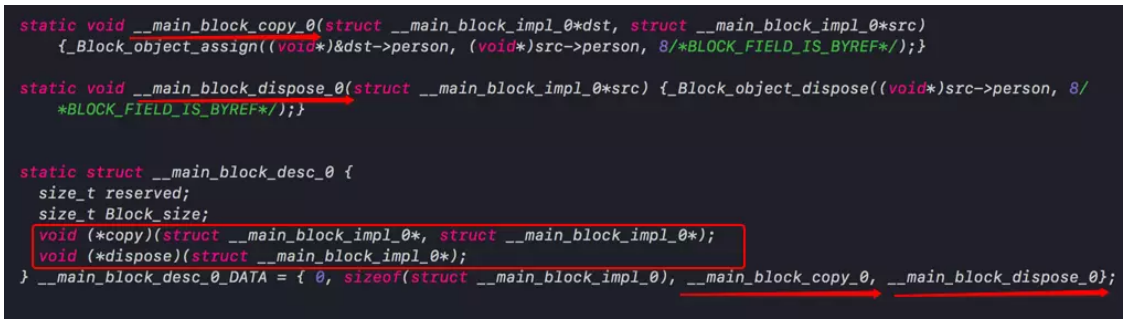
`xcrun -sdk iphoneos clang -arch arm64 -rewrite-objc -fobjc-arc -fobjc-runtime=ios-8.0.0 main.m`

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

__weak修饰的变量，在生成的`__main_block_impl_0`中也是使用`__weak`修饰。

__main_block_copy_0 和 __main_block_dispose_0

当block中捕获对象类型的变量时，我们发现block结构体`__main_block_impl_0`的描述结构体`__main_block_desc_0`中多了两个参数`copy`和`dispose`函数，查看源码：



`copy`和`dispose`函数中传入的都是`__main_block_impl_0`结构体本身。

> `copy`本质就是`__main_block_copy_0`函数，`__main_block_copy_0`函数内部调用`_Block_object_assign`函数，`_Block_object_assign`中传入的是person对象的地址，person对象，以及8。

> `dispose`本质就是`__main_block_dispose_0`函数，`__main_block_dispose_0`函数内部调用`_Block_object_dispose`函数，`_Block_object_dispose`函数传入的参数是person对象，以及8。

_Block_object_assign函数调用时机及作用

当block进行copy操作的时候就会自动调用`__main_block_desc_0`内部的`__main_block_copy_0`函数，`__main_block_copy_0`函数内部会调用`_Block_object_assign`函数。

`_Block_object_assign`函数会自动根据`__main_block_impl_0`结构体内部的`person`是什么类型的指针，对`person`对象产生强引用或者弱引用。可以理解为`_Block_object_assign`函数内部会对`person`进行引用计数器的操作，如果`__main_block_impl_0`结构体内`person`指针是`__strong`类型，则为强引用，引用计数+1，如果`__main_block_impl_0`结构体内`person`指针是`__weak`类型，则为弱引用，引用计数不变。

_Block_object_dispose函数调用时机及作用

当block从堆中移除时就会自动调用`__main_block_desc_0`中的`__main_block_dispose_0`函数，`__main_block_dispose_0`函数内部会调用`_Block_object_dispose`函数。

`_Block_object_dispose`会对`person`对象做释放操作，类似于`release`，也就是断开对`person`对象的引用，而`person`究竟是否被释放还是取决于`person`对象自己的引用计数。

总结

1. 一旦block中捕获的变量为对象类型，`block`结构体中的`__main_block_desc_0`会出两个参数`copy`和`dispose`。因为访问的是个对象，block希望拥有这个对象，就需要对对象进行引用，也就是进行内存管理的操作。比如说对对象进行retarn操作，因此一旦block捕获的变量是对象类型就会会自动生成`copy`和`dispose`来对内部引用的对象进行内存管理。

2. 当block内部访问了对象类型的auto变量时，如果block是在栈上，block内部不会对person产生强引用。不论block结构体内部的变量是`__strong`修饰还是`__weak`修饰，都不会对变量产生强引用。

3. 如果block被拷贝到堆上。`copy`函数会调用`_Block_object_assign`函数，根据auto变量的修饰符（__strong，__weak，unsafe_unretained）做出相应的操作，形成强引用或者弱引用

4. 如果block从堆中移除，`dispose`函数会调用`_Block_object_dispose`函数，自动释放引用的auto变量。

问题

1. 下列代码person在何时销毁？

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent
*)event
{
    Person *person = [[Person alloc] init];
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3.0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
        NSLog(@"%@",person);
    });
    NSLog(@"touchBegin----------End");
}
```

打印内容

[图片上传失败...(image-af8e7a-1528078840267)]

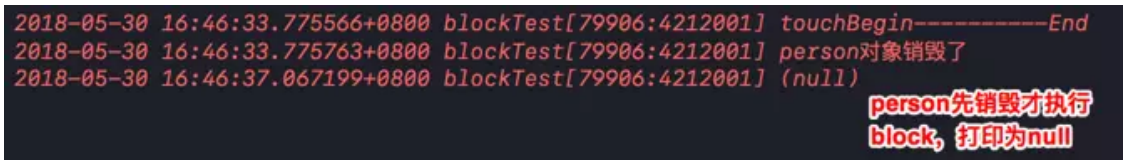
答：上文提到过RAC环境中，block作为GCD API的方法参数时会自动进行`copy`操作，因此`block`在堆空间，并且使用强引用访问`person`对象，因此`block`内

部 `copy` 函数会对 `person` 进行强引用。当 `block` 执行完毕需要被销毁时，调用 `dispose` 函数释放对 `person` 对象的引用，`person` 没有强指针指向时才会被销毁。

2. 下列代码person在何时销毁？

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent
*)event
{
    Person *person = [[Person alloc] init];

    __weak Person *waekP = person;
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3.0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
        NSLog(@"%@",waekP);
    });
    NSLog(@"touchBegin----------End");
}
```

打印内容



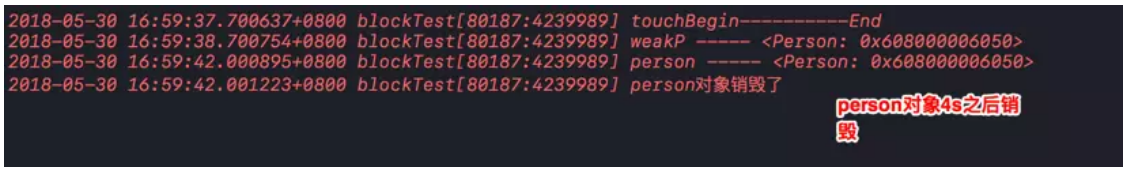
答：block中对 `waekP` 为 `__weak` 弱引用，因此 `block` 内部 `copy` 函数会对 `person` 同样进行弱引用，当大括号执行完毕时，`person` 对象没有强指针引用就会被释放。因此 `block` 块执行的时候打印 `null` 。

3. 通过示例代码进行总结。

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent
*)event
{
    Person *person = [[Person alloc] init];

    __weak Person *waekP = person;
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(1.0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{

        NSLog(@"weakP ----- %@",waekP);
```

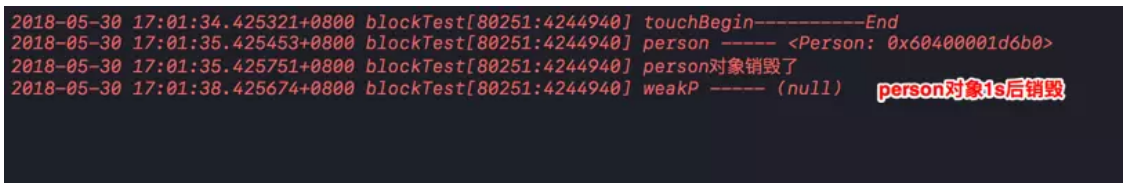
```
        dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)
(3.0 * NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
            NSLog(@"person ----- %@",person);
        });
    });
    NSLog(@"touchBegin----------End");
}
```

打印内容




```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent
*)event
{
    Person *person = [[Person alloc] init];

    __weak Person *waekP = person;
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(1.0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{

        NSLog(@"person ----- %@",person);

        dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3
* NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
            NSLog(@"weakP ----- %@",waekP);
        });
    });
    NSLog(@"touchBegin----------End");
}
```

打印内容



block内修改变量的值

本部分分析基于下面代码。

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int age = 10;
        Block block = ^ {
            // age = 20; // 无法修改
            NSLog(@"%d",age);
        };
        block();
    }
    return 0;
}
```

默认情况下block不能修改外部的局部变量。通过之前对源码的分析可以知道。

`age`是在`main`函数内部声明的，说明`age`的内存存在于`main`函数的栈空间内部，但是`block`内部的代码在`__main_block_func_0`函数内部。`__main_block_func_0`函数内部无法访问`age`变量的内存空间，两个函数的栈空间不一样，`__main_block_func_0`内部拿到的`age`是`block`结构体内部的`age`，因此无法在`__main_block_func_0`函数内部去修改`main`函数内部的变量。

方式一：age使用static修饰。

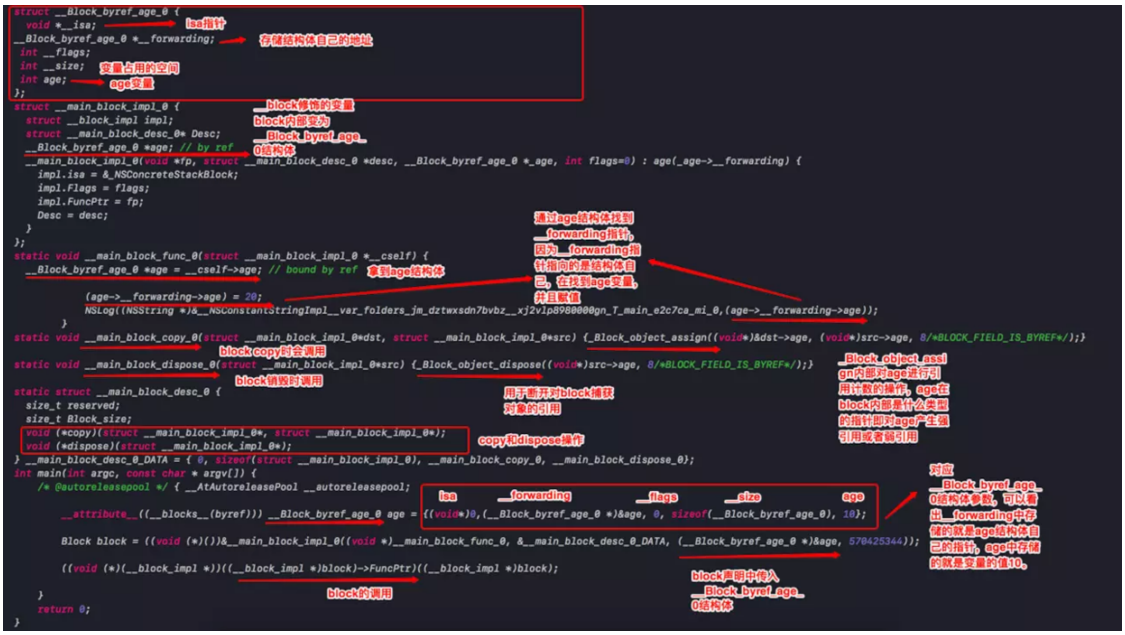
前文提到过static修饰的`age`变量传递到block内部的是指针，在`__main_block_func_0`函数内部就可以拿到`age`变量的内存地址，因此就可以在block内部修改age的值。

方式二：__block

__block用于解决block内部不能修改auto变量值的问题，__block不能修饰静态变量（static） 和全局变量

```
__block int age = 10;
```

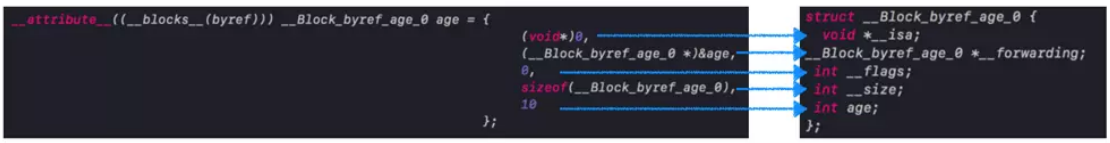
编译器会将__block修饰的变量包装成一个对象，查看其底层c++源码。



上述源码中可以发现

首先被 `__block` 修饰的 `age` 变量声明变为名为 `age` 的 `__Block_byref_age_0` 结构体，也就是说加上 `__block` 修饰的话捕获到的 `block` 内的变量为 `__Block_byref_age_0` 类型的结构体。

通过下图查看 `__Block_byref_age_0` 结构体内存储哪些元素。



`__isa指针` ： `__Block_byref_age_0` 中也有isa指针也就是说 `__Block_byref_age_0` 本质也一个对象。

`__forwarding` ： `__forwarding` 是 `__Block_byref_age_0` 结构体类型的，并且 `__forwarding` 存储的值为 `(__Block_byref_age_0 *)&age` ，即结构体自己的内存地址。

`__flags` ： 0

`__size` ： `sizeof(__Block_byref_age_0)` 即 `__Block_byref_age_0` 所占用的内存

空间。

`age` ： 真正存储变量的地方，这里存储局部变量10。

接着将 `__Block_byref_age_0` 结构体 `age` 存入 `__main_block_impl_0` 结构体中，并赋值给 `__Block_byref_age_0 *age;`

```
Block block = ((void (*)())&__main_block_impl_0(
                                                (void *)__main_block_func_0,
                                                &__main_block_desc_0_DATA,
                                                (__Block_byref_age_0 *)&age,
                                                570425344)
                                               );
```

```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  __Block_byref_age_0 *age; // by ref
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, __Block_byref_age_0 *_age, int flags=0) : age(_age->__forwarding) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
```

之后调用 `block` ，首先取出 `__main_block_impl_0` 中的 `age` ，通过age结构体拿到 `__forwarding` 指针，上面提到过 `__forwarding中` 保存的就是 `__Block_byref_age_0` 结构体本身，这里也就是 `age(__Block_byref_age_0)` ，在通过 `__forwarding` 拿到结构体中的 `age(10)` 变量并修改其值。

后续NSLog中使用 `age` 时也通过同样的方式获取 `age` 的值。

```
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
  __Block_byref_age_0 *age = __cself->age; // bound by ref 拿到age结构体
                                           通过__forwading修
        (age->__forwarding->age) = 20; 改结构体内变量age值
        NSLog((NSString *)&__NSConstantStringImpl__var_folders_jm_dztwxsdn7bvbz__xj2vlp8980000gn_T_main_e2c7ca_mi_0,(age->__forwarding->age));
    }
```

为什么要通过__forwarding获取age变量的值?

`__forwarding` 是指向自己的指针。这样的做法是为了方便内存管理，之后内存管理章节会详细解释。

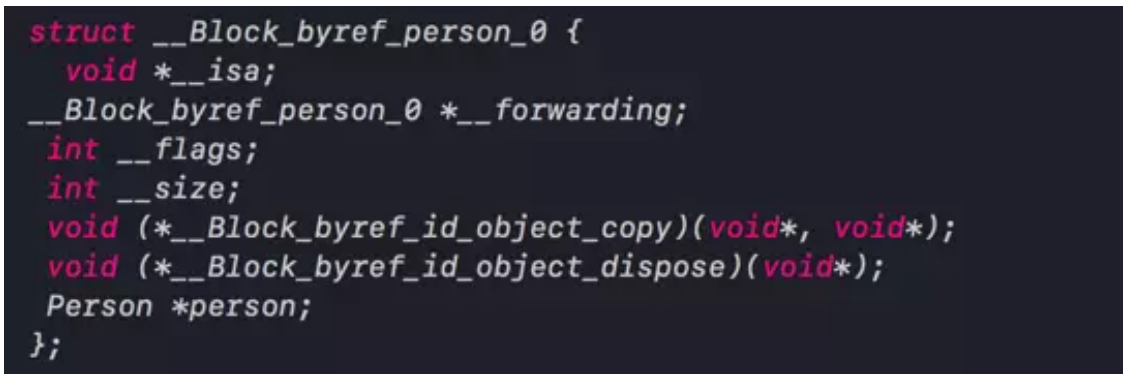
到此为止， `__block` 为什么能修改变量的值已经很清晰了。 `__block` 将变量包装成对象，然后在把 `age` 封装在结构体里面，block内部存储的变量为结构体指针，也就可以通过指针找到内存地址进而修改变量的值。

__block修饰对象类型

那么如果变量本身就是对象类型呢？通过以下代码生成c++源码查看

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        __block Person *person = [[Person alloc] init];
        NSLog(@"%@",person);
        Block block = ^{
            person = [[Person alloc] init];
            NSLog(@"%@",person);
        };
        block();
    }
    return 0;
}
```

通过源码查看，将对象包装在一个新的结构体中。结构体内部会有一个`person`对象，不一样的地方是结构体内部添加了内存管理的两个函数`__Block_byref_id_object_copy`和`__Block_byref_id_object_dispose`



```
struct __Block_byref_person_0 {
  void *__isa;
__Block_byref_person_0 *__forwarding;
 int __flags;
 int __size;
 void (*__Block_byref_id_object_copy)(void*, void*);
 void (*__Block_byref_id_object_dispose)(void*);
 Person *person;
};
```

`__Block_byref_id_object_copy`和`__Block_byref_id_object_dispose`函数的调用时机及作用在__block内存管理部分详细分析。

问题

1. 以下代码是否可以正确执行

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSMutableArray *array = [NSMutableArray array];
        Block block = ^{
            [array addObject: @"5"];
            [array addObject: @"5"];
            NSLog(@"%@",array);
        };
```

```
            block();
        }
        return 0;
    }
```

答：可以正确执行，因为在block块中仅仅是使用了array的内存地址，往内存地址中添加内容，并没有修改arry的内存地址，因此array不需要使用__block修饰也可以正确编译。

因此当仅仅是使用局部变量的内存地址，而不是修改的时候，尽量不要添加__block，通过上述分析我们知道一旦添加了__block修饰符，系统会自动创建相应的结构体，占用不必要的内存空间。

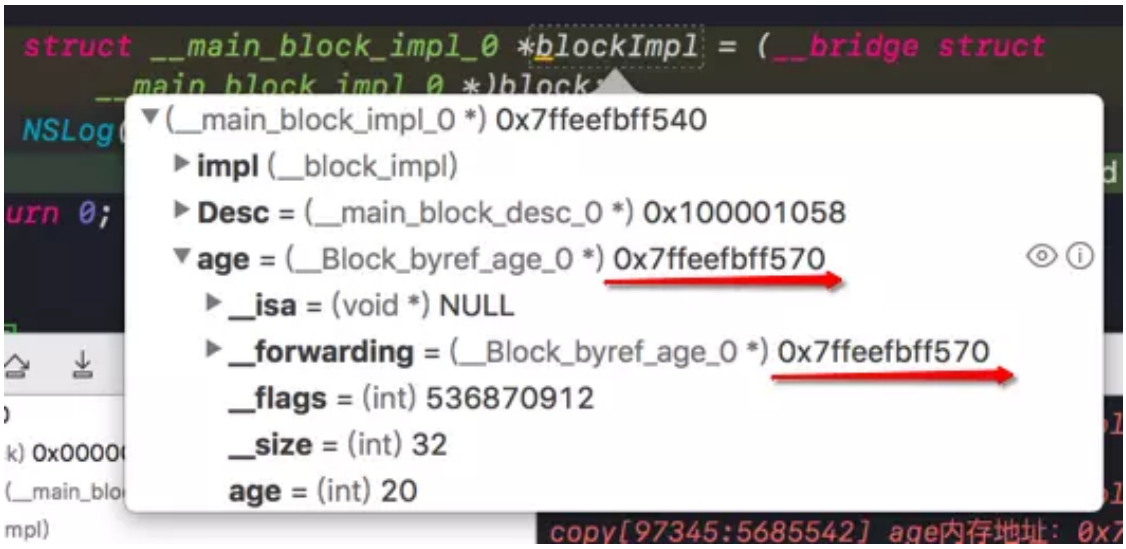
2. 上面提到过 `__block` 修饰的 `age` 变量在编译时会被封装为结构体，那么当在外部使用 `age` 变量的时候，使用的是 `__Block_byref_age_0` 结构体呢？还是 `__Block_byref_age_0` 结构体内的age变量呢？

为了验证上述问题 同样使用自定义结构体的方式来查看其内部结构

```
typedef void (^Block)(void);

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

struct __main_block_desc_0 {
    size_t reserved;
    size_t Block_size;
    void (*copy)(void);
    void (*dispose)(void);
};

struct __Block_byref_age_0 {
    void *__isa;
    struct __Block_byref_age_0 *__forwarding;
    int __flags;
    int __size;
    int age;
};
struct __main_block_impl_0 {
```

```
    struct __block_impl impl;
    struct __main_block_desc_0* Desc;
    struct __Block_byref_age_0 *age; // by ref
};

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        __block int age = 10;
        Block block = ^{
            age = 20;
            NSLog(@"age is %d",age);
        };
        block();
        struct __main_block_impl_0 *blockImpl = (__bridge struct
__main_block_impl_0 *)block;
        NSLog(@"%p",&age);
    }
    return 0;
}
```

打印断点查看结构体内部结构



通过查看blockImpl结构体其中的内容，找到`age`结构体，其中重点观察两个元素：

1. `__forwarding`其中存储的地址确实是age结构体变量自己的地址
2. `age`中存储这修改后的变量20。

上面也提到过，在`block`中使用或修改`age`的时候都是通过结构体`__Block_byref_age_0`找到`__forwarding`在找到变量`age`的。

另外apple为了隐藏 `__Block_byref_age_0` 结构体的实现，打印age变量的地址发现其实是 `__Block_byref_age_0` 结构体内 `age` 变量的地址。

通过上图的计算可以发现打印 `age` 的地址同 `__Block_byref_age_0` 结构体内 `age` 值的地址相同。也就是说外面使用的age，代表的就是结构体内的age值。所以直接拿来用的 `age` 就是之前声明的 `int age` 。

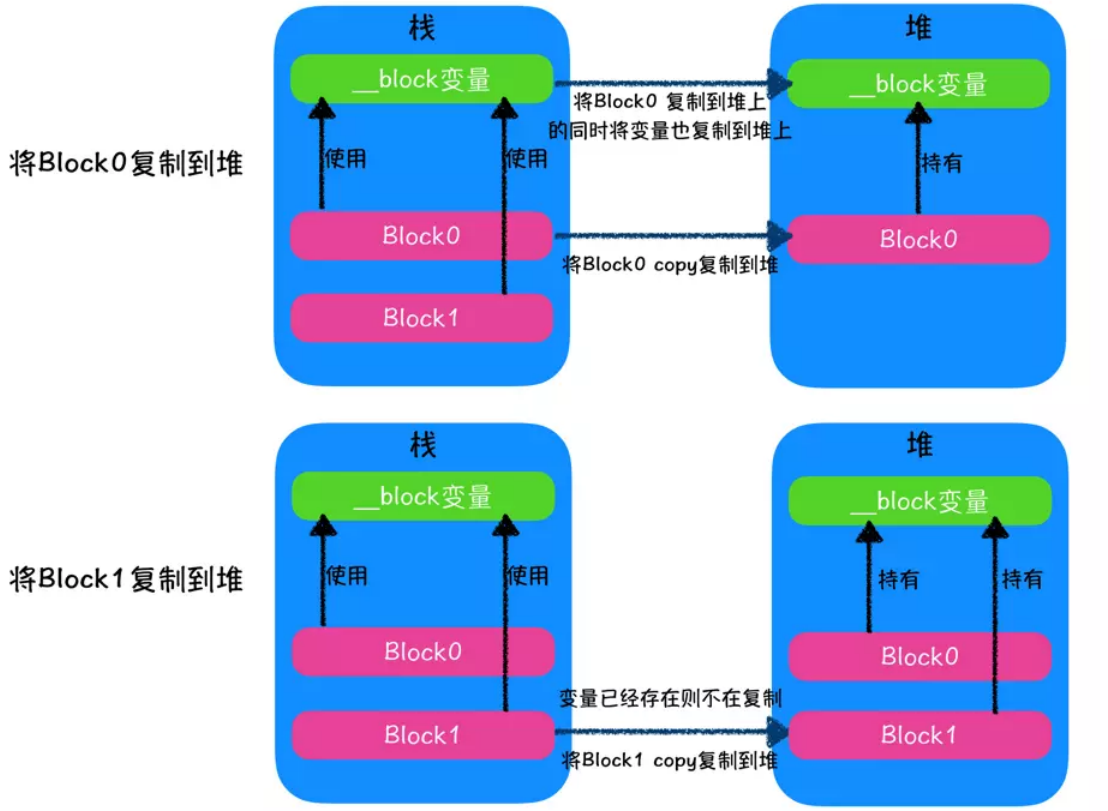
__block内存管理

上文提到当block中捕获对象类型的变量时，block中的 `__main_block_desc_0` 结构体内部会自动添加 `copy` 和 `dispose` 函数对捕获的变量进行内存管理。

那么同样的当block内部捕获 `__block` 修饰的对象类型的变量时， `__Block_byref_person_0` 结构体内部也会自动添加 `__Block_byref_id_object_copy` 和 `__Block_byref_id_object_dispose` 对被 `__block` 包装成结构体的对象进行内存管理。

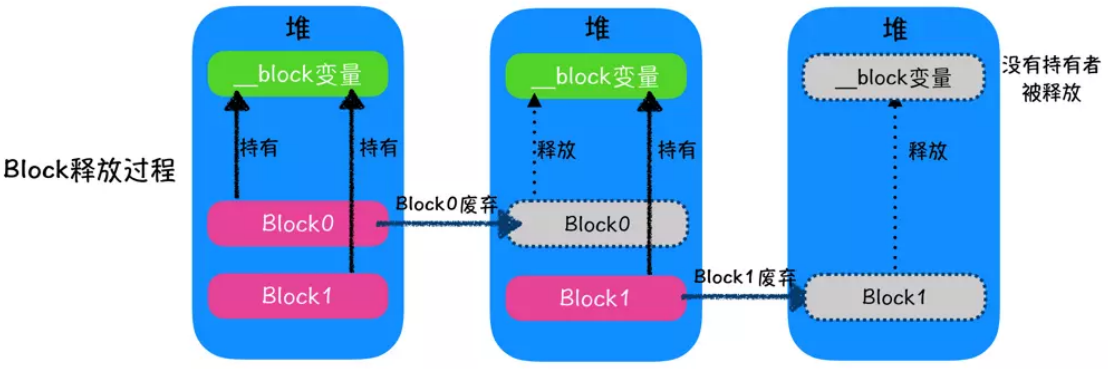
当 `block` 内存在栈上时，并不会对 `__block` 变量产生内存管理。
当 `blcok` 被 `copy` 到堆上时 会调用 `block` 内部的 `copy` 函数， `copy` 函数内部会调用 `_Block_object_assign` 函数， `_Block_object_assign` 函数会对 `__block` 变量形成强引用（相当于retain）

首先通过一张图看一下block复制到堆上时内存变化



当 `block` 被 `copy` 到堆上时， `block` 内部引用的 `__block` 变量也会被复制到堆上，并且持有变量，如果 `block` 复制到堆上的同时， `__block` 变量已经存在堆上了，则不会复制。

当block从堆中移除的话，就会调用dispose函数，也就是 `__main_block_dispose_0` 函数， `__main_block_dispose_0` 函数内部会调用 `_Block_object_dispose` 函数，会自动释放引用的__block变量。



block内部决定什么时候将变量复制到堆中，什么时候对变量做引用计数的操作。

`__block` 修饰的变量在block结构体中一直都是强引用，而其他类型的是由传入的对象指针类型决定。

一段代码更深入的观察一下。

```
typedef void (^Block)(void);
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int number = 20;
        __block int age = 10;

        NSObject *object = [[NSObject alloc] init];
        __weak NSObject *weakObj = object;

        Person *p = [[Person alloc] init];
        __block Person *person = p;
        __block __weak Person *weakPerson = p;

        Block block = ^ {
            NSLog(@"%d",number); // 局部变量
            NSLog(@"%d",age); // __block修饰的局部变量
            NSLog(@"%p",object); // 对象类型的局部变量
            NSLog(@"%p",weakObj); // __weak修饰的对象类型的局部变量
            NSLog(@"%p",person); // __block修饰的对象类型的局部变量
            NSLog(@"%p",weakPerson); // __block，__weak修饰的对象类型的
局部变量
        };
        block();
    }
    return 0;
}
```

将上述代码转化为c++代码查看不同变量之间的区别

```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;

  int number;
  NSObject *__strong object;
  NSObject *__weak weakObj;
  __Block_byref_age_0 *age; // by ref
  __Block_byref_person_1 *person; // by ref
```

```
  __Block_byref_weakPerson_2 *weakPerson; // by ref

  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc,
int _number, NSObject *__strong _object, NSObject *__weak _weakObj,
__Block_byref_age_0 *_age, __Block_byref_person_1 *_person,
__Block_byref_weakPerson_2 *_weakPerson, int flags=0) :
number(_number), object(_object), weakObj(_weakObj), age(_age-
>__forwarding), person(_person->__forwarding),
weakPerson(_weakPerson->__forwarding) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
```

上述 `__main_block_impl_0` 结构体中看出，没有使用 `__block` 修饰的变量（object 和 weadObj）则根据他们本身被block捕获的指针类型对他们进行强引用或弱引用，而一旦使用 `__block` 修饰的变量， `__main_block_impl_0` 结构体内一律使用强指针引用生成的结构体。

接着我们来看 `__block` 修饰的变量生成的结构体有什么不同

```
struct __Block_byref_age_0 {
  void *__isa;
__Block_byref_age_0 *__forwarding;
 int __flags;
 int __size;
 int age;
};

struct __Block_byref_person_1 {
  void *__isa;
__Block_byref_person_1 *__forwarding;
 int __flags;
 int __size;
 void (*__Block_byref_id_object_copy)(void*, void*);
 void (*__Block_byref_id_object_dispose)(void*);
 Person *__strong person;
};

struct __Block_byref_weakPerson_2 {
  void *__isa;
__Block_byref_weakPerson_2 *__forwarding;
```

```
 int __flags;
 int __size;
 void (*__Block_byref_id_object_copy)(void*, void*);
 void (*__Block_byref_id_object_dispose)(void*);
 Person *__weak weakPerson;
};
```

如上面分析的那样，`__block`修饰对象类型的变量生成的结构体内部多了`__Block_byref_id_object_copy`和`__Block_byref_id_object_dispose`两个函数，用来对对象类型的变量进行内存管理的操作。而结构体对对象的引用类型，则取决于block捕获的对象类型的变量。`weakPerson`是弱指针，所以`__Block_byref_weakPerson_2`对`weakPerson`就是弱引用，`person`是强指针，所以`__Block_byref_person_1`对person就是强引用。

```
static void __main_block_copy_0(struct __main_block_impl_0*dst,
struct __main_block_impl_0*src) {
    _Block_object_assign((void*)&dst->age, (void*)src->age,
8/*BLOCK_FIELD_IS_BYREF*/);
    _Block_object_assign((void*)&dst->object, (void*)src->object,
3/*BLOCK_FIELD_IS_OBJECT*/);
    _Block_object_assign((void*)&dst->weakObj, (void*)src->weakObj,
3/*BLOCK_FIELD_IS_OBJECT*/);
    _Block_object_assign((void*)&dst->person, (void*)src->person,
8/*BLOCK_FIELD_IS_BYREF*/);
    _Block_object_assign((void*)&dst->weakPerson, (void*)src-
>weakPerson, 8/*BLOCK_FIELD_IS_BYREF*/);
}
```

`__main_block_copy_0`函数中会根据变量是强弱指针及有没有被`__block`修饰做出不同的处理，强指针在block内部产生强引用，弱指针在block内部产生弱引用。被`__block`修饰的变量最后的参数传入的是8，没有被`__block`修饰的变量最后的参数传入的是3。

当block从堆中移除时通过dispose函数来释放他们。

```
static void __main_block_dispose_0(struct __main_block_impl_0*src)
{
    _Block_object_dispose((void*)src->age,
8/*BLOCK_FIELD_IS_BYREF*/);
    _Block_object_dispose((void*)src->object,
```

```
3/*BLOCK_FIELD_IS_OBJECT*/);
    _Block_object_dispose((void*)src->weakObj,
3/*BLOCK_FIELD_IS_OBJECT*/);
    _Block_object_dispose((void*)src->person,
8/*BLOCK_FIELD_IS_BYREF*/);
    _Block_object_dispose((void*)src->weakPerson,
8/*BLOCK_FIELD_IS_BYREF*/);

}
```

__forwarding指针

上面提到过 `__forwarding` 指针指向的是结构体自己。当使用变量的时候，通过结构体找到 `__forwarding` 指针，在通过 `__forwarding` 指针找到相应的变量。这样设计的目的是为了方便内存管理。通过上面对 `__block` 变量的内存管理分析我们知道， `block` 被复制到堆上时，会将 `block` 中引用的变量也复制到堆中。

我们重回到源码中。当在block中修改 `__block` 修饰的变量时。

```
static void __main_block_func_0(struct __main_block_impl_0
*__cself) {
  __Block_byref_age_0 *age = __cself->age; // bound by ref
            (age->__forwarding->age) = 20;
            NSLog((NSString
*)&__NSConstantStringImpl__var_folders_jm_dztwxsdn7bvbz__xj2vlp8980
000gn_T_main_b05610_mi_0,(age->__forwarding->age));
        }
```

通过源码可以知道，当修改 `__block` 修饰的变量时，是根据变量生成的结构体这里是 `__Block_byref_age_0` 找到其中 `__forwarding` 指针， `__forwarding` 指针指向的是结构体自己因此可以找到age变量进行修改。

当block在栈中时， `__Block_byref_age_0` 结构体内的 `__forwarding` 指针指向结构体自己。

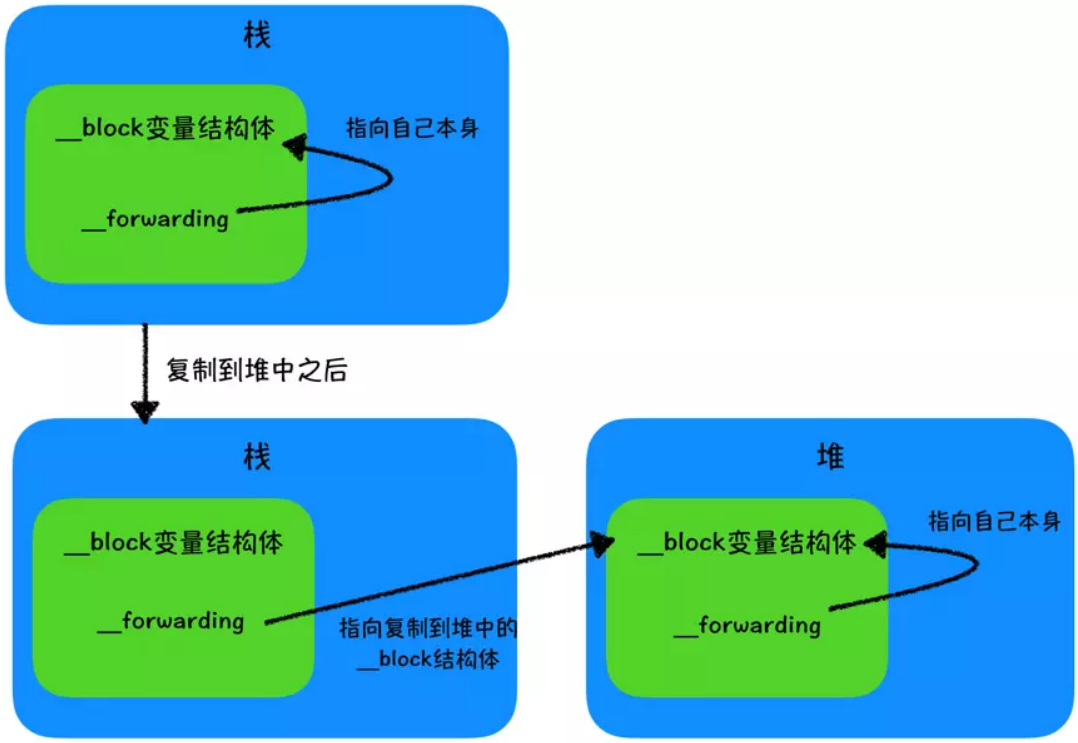
而当block被复制到堆中时，栈中的 `__Block_byref_age_0` 结构体也会被复制到堆中一份，而此时栈中的 `__Block_byref_age_0` 结构体中的 `__forwarding` 指针指向的就是堆中的 `__Block_byref_age_0` 结构体，堆中 `__Block_byref_age_0` 结构体内的 `__forwarding` 指针依然指向自己。

此时当对age进行修改时

```
// 栈中的age
__Block_byref_age_0 *age = __cself->age; // bound by ref
// age->__forwarding获取堆中的age结构体
// age->__forwarding->age 修改堆中age结构体的age变量
(age->__forwarding->age) = 20;
```

通过 `__forwarding` 指针巧妙的将修改的变量赋值在堆中的 `__Block_byref_age_0` 中。

我们通过一张图展示 `__forwarding` 指针的作用



因此block内部拿到的变量实际就是在堆上的。当block进行copy被复制到堆上时， `_Block_object_assign` 函数内做的这一系列操作。

被__block修饰的对象类型的内存管理

使用以下代码，生成c++代码查看内部实现

```
typedef void (^Block)(void);
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        __block Person *person = [[Person alloc] init];
        Block block = ^ {
            NSLog(@"%p", person);
        };
        block();
    }
    return 0;
}
```

来到源码查看 `__Block_byref_person_0` 结构体及其声明

```
__Block_byref_person_0结构体

typedef void (*Block)(void);
struct __Block_byref_person_0 {
  void *__isa;  // 8 内存空间
__Block_byref_person_0 *__forwarding; // 8
 int __flags; // 4
 int __size;  // 4
 void (*__Block_byref_id_object_copy)(void*, void*); // 8
 void (*__Block_byref_id_object_dispose)(void*); // 8
 Person *__strong person; // 8
};
// 8 + 8 + 4 + 4 + 8 + 8 + 8 = 48
```

```
// __Block_byref_person_0结构体声明

__attribute__((__blocks__(byref))) __Block_byref_person_0 person =
{
    (void*)0,
    (__Block_byref_person_0 *)&person,
    33554432,
    sizeof(__Block_byref_person_0),
    __Block_byref_id_object_copy_131,
    __Block_byref_id_object_dispose_131,

    ((Person *(*)(id, SEL))(void *)objc_msgSend)((id)((Person *(*)
(id, SEL))(void *)objc_msgSend)((id)objc_getClass("Person"),
```

```
  sel_registerName("alloc")), sel_registerName("init"))
  };
```

之前提到过 `__block` 修饰的对象类型生成的结构体中新增加了两个函数 `void (*__Block_byref_id_object_copy)(void*, void*);` 和 `void (*__Block_byref_id_object_dispose)(void*);` 。这两个函数为 `__block` 修饰的对象提供了内存管理的操作。

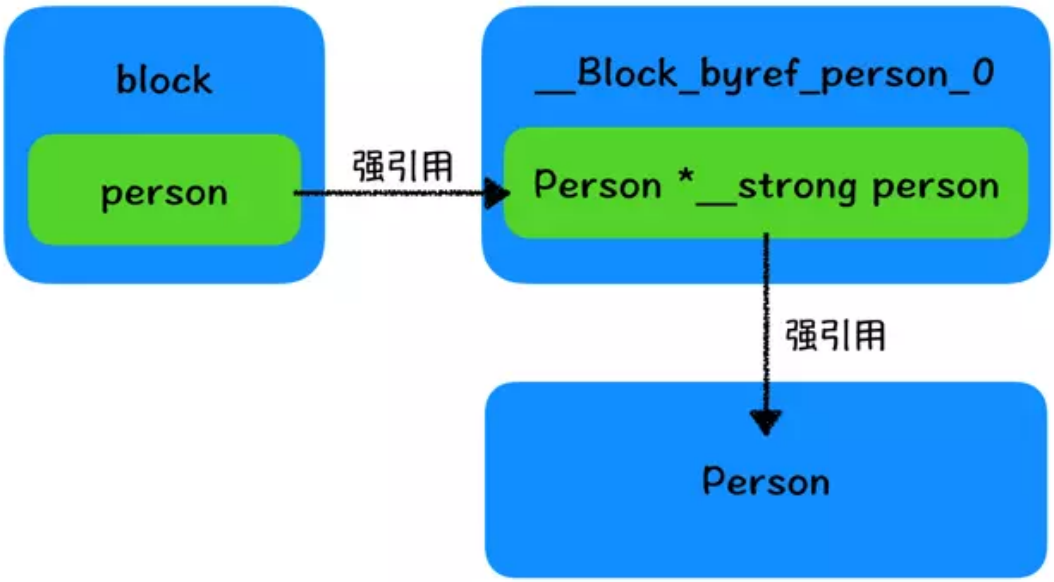
可以看出为 `void (*__Block_byref_id_object_copy)(void*, void*);` 和 `void (*__Block_byref_id_object_dispose)(void*);` 赋值的分别为 `__Block_byref_id_object_copy_131` 和 `__Block_byref_id_object_dispose_131` 。找到这两个函数

```
static void __Block_byref_id_object_copy_131(void *dst, void *src)
{
 _Block_object_assign((char*)dst + 40, *(void * *) ((char*)src + 40), 131);
}
static void __Block_byref_id_object_dispose_131(void *src) {
 _Block_object_dispose(*(void * *) ((char*)src + 40), 131);
}
```

上述源码中可以发现 `__Block_byref_id_object_copy_131` 函数中同样调用了 `_Block_object_assign` 函数，而 `_Block_object_assign` 函数内部拿到 `dst` 指针即 `block` 对象自己的地址值加上40个字节。并且 `_Block_object_assign` 最后传入的参数是131，同block直接对对象进行内存管理传入的参数3，8都不同。可以猜想 `_Block_object_assign` 内部根据传入的参数不同进行不同的操作的。

通过对上面 `__Block_byref_person_0` 结构体占用空间计算发现 `__Block_byref_person_0` 结构体占用的空间为48个字节。而加40恰好指向的就为 `person` 指针。

也就是说copy函数会将person地址传入 `_Block_object_assign` 函数， `_Block_object_assign` 中对Person对象进行强引用或者弱引用。



如果使用__weak修饰变量查看一下其中的源码

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person = [[Person alloc] init];
        __block __weak Person *weakPerson = person;
        Block block = ^ {
            NSLog(@"%p", weakPerson);
        };
        block();
    }
    return 0;
}
```

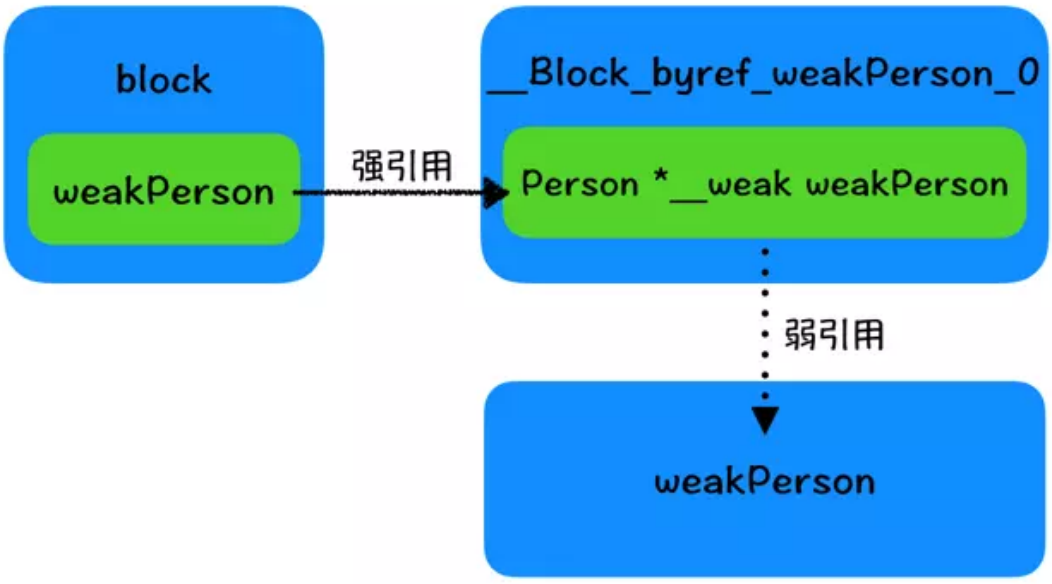
```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  __Block_byref_weakPerson_0 *weakPerson; // by ref
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc,
__Block_byref_weakPerson_0 *_weakPerson, int flags=0) :
weakPerson(_weakPerson->__forwarding) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
```

```
  }
};
```

`__main_block_impl_0` 中没有任何变化， `__main_block_impl_0` 对 `weakPerson` 依然是强引用，但是 `__Block_byref_weakPerson_0` 中对 `weakPerson` 变为了 `__weak` 指针。

```
struct __Block_byref_weakPerson_0 {
  void *__isa;
__Block_byref_weakPerson_0 *__forwarding;
 int __flags;
 int __size;
 void (*__Block_byref_id_object_copy)(void*, void*);
 void (*__Block_byref_id_object_dispose)(void*);
 Person *__weak weakPerson;
};
```

也就是说无论如何 `block` 内部中对 `__block` 修饰变量生成的结构体都是强引用，结构体内部对外部变量的引用取决于传入block内部的变量是强引用还是弱引用。



mrc环境下，尽管调用了copy操作， `__block` 结构体不会对 `person` 产生强引用，依然是弱引用。

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        __block Person *person = [[Person alloc] init];
        Block block = [^ {
            NSLog(@"%p", person);
        } copy];
        [person release];
        block();
        [block release];
    }
    return 0;
}
```

上述代码person会先释放

```
block的copy[50480:8737001] -[Person dealloc]
block的copy[50480:8737001] 0x100669a50
```

当block从堆中移除的时候。会调用`dispose`函数，block块中去除对`__Block_byref_person_0 *person;`的引用，`__Block_byref_person_0`结构体中也会调用`dispose`操作去除对`Person *person;`的引用。以保证结构体和结构体内部的对象可以正常释放。

循环引用

循环引用导致内存泄漏。

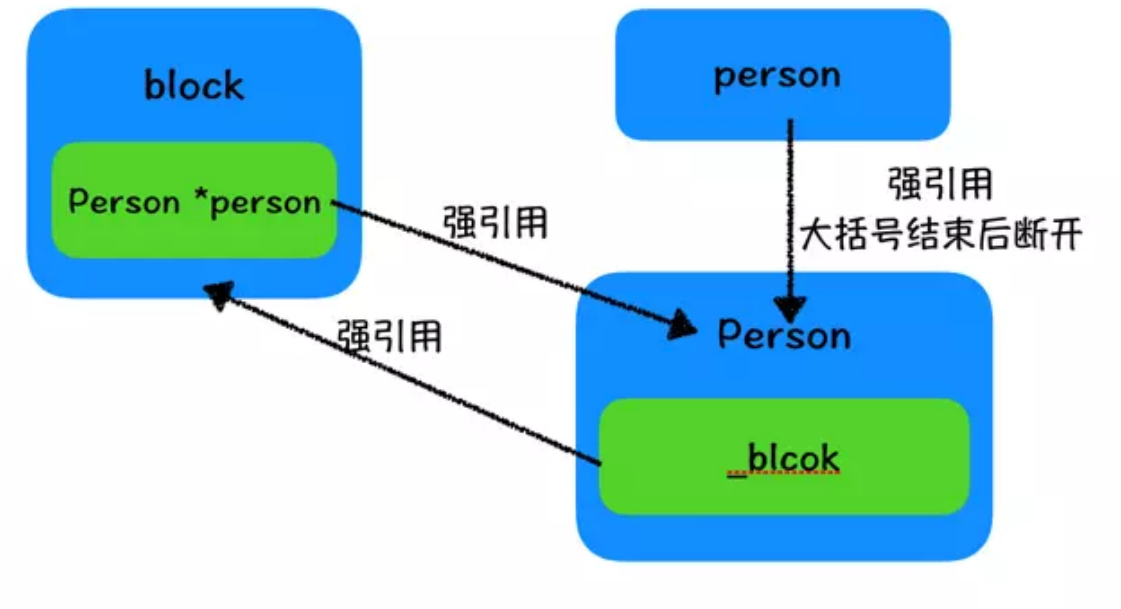
```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person = [[Person alloc] init];
        person.age = 10;
        person.block = ^{
            NSLog(@"%d",person.age);
        };
    }
    NSLog(@"大括号结束啦");
    return 0;
}
```

运行代码打印内容

```
block的copy[55423:9158212] 大括号结束啦
```

可以发现大括号结束之后，`person`依然没有被释放，产生了循环引用。

通过一张图看一下他们之间的内存结构



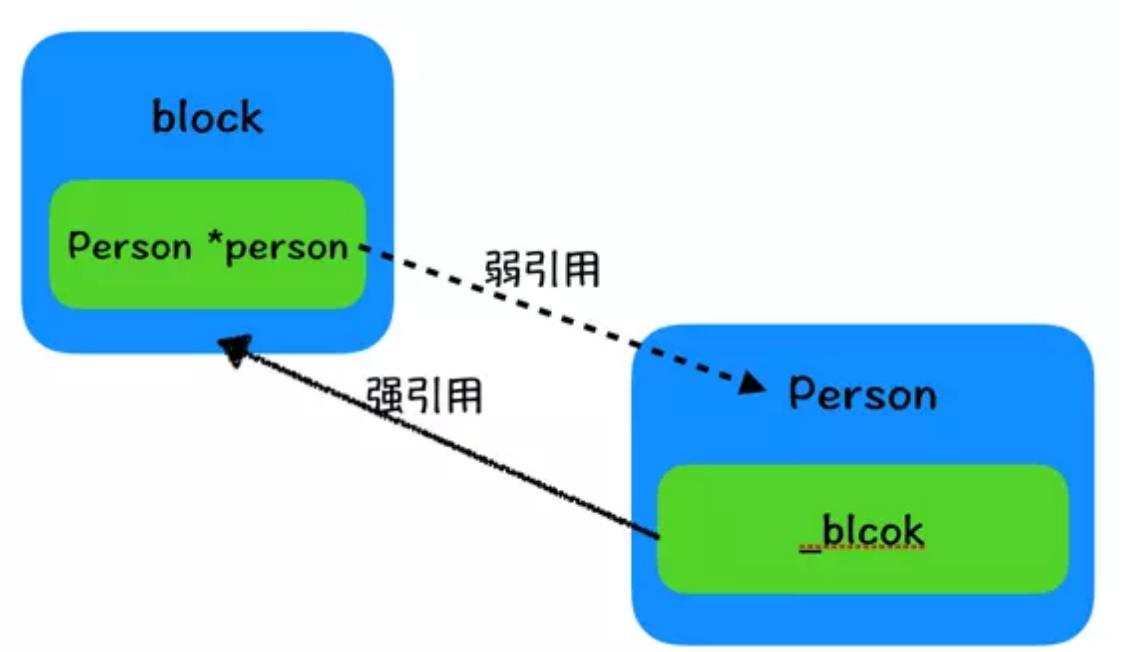
上图中可以发现，Person对象和block对象相互之间产生了强引用，导致双方都不会被释放，进而造成内存泄漏。

解决循环引用问题 - ARC

首先为了能随时执行block，我们肯定希望`person`对block对强引用，而block内部对`person`的引用为弱引用最好。

使用`__weak`和`__unsafe_unretained`修饰符可以解决循环引用的问题

我们上面也提到过`__weak`会使`block`内部将指针变为弱指针。`block`对`person`对象为弱指针的话，也就不会出现相互引用而导致不会被释放了。



`__weak` 和 `__unsafe_unretained` 的区别。

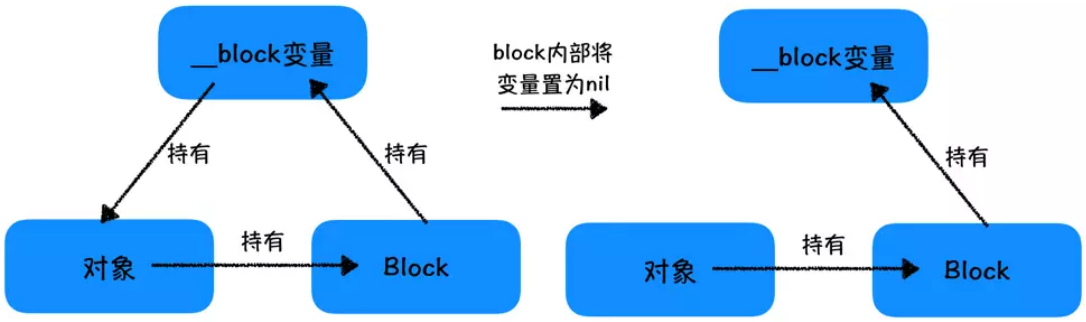
`__weak` 不会产生强引用，指向的对象销毁时，会自动将指针置为nil。因此一般通过 `__weak` 来解决问题。

`__unsafe_unretained` 不会产生前引用，不安全，指向的对象销毁时，指针存储的地址值不变。

使用 `__block` 也可以解决循环引用的问题。

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        __block Person *person = [[Person alloc] init];
        person.age = 10;
        person.block = ^{
            NSLog(@"%d",person.age);
            person = nil;
        };
        person.block();
    }
    NSLog(@"大括号结束啦");
    return 0;
}
```

上述代码之间的相互引用可以使用下图表示



上面我们提到过，在block内部使用变量使用的其实是 `__block` 修饰的变量生成的结构体 `__Block_byref_person_0` 内部的 `person` 对象，那么当 `person` 对象置为nil也就断开了结构体对person的强引用，那么三角的循环引用就自动断开。该释放的时候就会释放了。但是有弊端，必须执行block，并且在block内部将 `person` 对象置为nil。也就是说在block执行之前代码是因为循环引用导致内存泄漏的。

解决循环引用问题 - MRC

使用 `__unsafe_unretained` 解决。在MRC环境下不支持使用 `__weak` ，使用原理同ARC环境下相同，这里不在赘述。

使用 `__block` 也能解决循环引用的问题。因为上文 `__block` 内存管理中提到过，MRC环境下，尽管调用了copy操作， `__block` 结构体不会对person产生强引用，依然是弱引用。因此同样可以解决循环引用的问题。

`__strong` 和 `__weak`

```
__weak typeof(self) weakSelf = self;
person.block = ^{
    __strong typeof(weakSelf) myself = weakSelf;
    NSLog(@"age is %d", myself->_age);
};
```

在 `block` 内部重新使用 `__strong` 修饰 `self` 变量是为了在 `block` 内部有一个强指针指向 `weakSelf` 避免在 `block` 调用的时候 `weakSelf` 已经被销毁。