Managing Build Settings and Avoiding pbxproj Conflicts

Merge conflicts are not fun to deal with, sometimes they can be downright nasty to resolve. This can be especially true when they happen on the `project.pbxproj` files. Conflicts on this file are more tricky to resolve than most because it governs the contents of your project and actions taken during the build process. This can be non-trivial to deal with on simple project files, and downright crushing when it happens on complex projects. As part of my job I maintain our project files and build systems. This post is about some of the methods I use to avoid causing file conflicts on `pbxproj` files altogether.

Target Management and Cleanliness

Within the `project.pbxproj` file there is a root object, this defines the base structure and layout of the contents of the whole file. This root object defines the way the files are organized in Xcode's Navigator panel, as well as a list of targets that are defined in the project. The more targets you have per project the more files you will have in total, and if any of those files are used in multiple targets the greater the likelihood of a merge conflict occuring.

Avoid Multiple Targets for Building the Same Product

The most major thing to avoid is having multiple targets that build the same product in slightly different ways. This means you should avoid having a target for each platform (OS X, iOS, WatchOS, tvOS) you want to build for. Having multiple targets that share the same files and settings within the `project.pbxproj` file is a recipe for disaster. Since each of these files are identifier by a unique identifier, bad merges can happen that will turn your project file to garbage due to some unfortunately matching logic on these identifier strings. If you need to build a framework or library for multiple platforms, then you should be using schemes with different build configurations set to alter the values of your build settings to do this rather than duplicate the target multiple times. This is the purpose for scheme and build configuration

pairings within the Xcode build system. Each build configuration allows an xcconfig file to be set per target to import custom settings to use. This is where you should be storing the platform-specific values for building the target. By removing duplicated targets you drastically cut down on the footprint of the file and amount of places where a bad merge could happen and break your project file.

Avoid Massive Targets

All applications start out small, then will grow in size over time. Many people will just keep adding new files to their application target because it is a very simple process. After a while this becomes completely unmanageable and difficult to change. If this sounds like your app, then you should consider taking an intitial step of grouoping code into separate targets that better describe the functionality they add or perform for the app. Slimming down the size of the targets will make it easier to determine if `ViewControllerFoo.m` should not be include in the `CoreDataModels.a` target and that line of your conflict can be removed. Making many smaller targets will also give you the benefit of speeding up build times by allowing these independent bits of code be compiled in parallel.

Number of Targets per Project

In the last section I mentioned that very large targets are not advisble. My other suggestion about target management is to avoid creating any more than one or two targets per project file (this is not including test targets). This may sound like contradictory advice, but this is a two step process; 1. Make smaller and more modular targets, 2. move these small targets into separate project files. This keeps each project file only referencing files it absolutely needs and a tight focus on what the purpose of the project file is. This can result in the use of Xcode Workspaces over Xcode Projects due to the need to be able to build targets from many project files at once. Once you have done this step in management, you will want to add some test targets to each of your new project files to accompany the executable targets that were created. This will reduce the possibility of merge conflicts because it makes it much more unlikely that any one of these project files will be altered in any sigificant way in any one commit.

Target Hygene

Another thing that should be done is some regularly hygene maintenance of your project files and targets. This means periodically going through and

cleaning out erronously added build settings, removing old unused code from targets and remove the references to those files from the project or delete them entirely. To make the changes be merged easier, any time you make any sort of modificiation to your project files you should make each edit a separate commit. This is extremely good practice because Xcode lacks any sort of "undo" mechanism for changes made to a project file's configuration. Without making a commit on each small change you can -- it can lead to a single mistake costing your hours of time trying to restore the state of the project file. Xcode has a built in way of managing this, called Xcode Snapshots, but I have seen so few people use this or understand why they would want to that I recommend people directly commit these changes to source control instead. This means any time a `pbxproj` file shows up in a modified state in source control, you should make a separate commit that only contain the `project.pbxproj` file to make tracking the state changes easier.

Schemes and Build Configurations

So far all the suggestions have been centered around how to manage the changes to the `project.pbxproj` file and how to reduce the overall size and contents of that file. Earlier in the post I mentioned the use of Xcode Schemes and Build Configurations. It is important to point out that Schemes exist outside of the project file, and contain references that are resolved to identifiers inside of the project file. There is no overhead in creating and using many different schemes. Doing this will not result in your project file being harder to manage. Build Configurations on the other hand, are stored within the project file. Each Build Configuration contains a list of build settings that are used to influence how the build is performed. By default Xcode adds a lot of default settings to each target when it is created. Many of these are helpful, but this can become extremely burdensome to maintain so it is advisable to use xcconfig files to manage this instead.

Build Settings Management

There is another step that can be taken that can drastically reduce the file size of your Xcode Project files, build setting management. Not only does each target in your project file have a set of build settings, but also the project file has a set of build settings that it uses as a base configuration.

`xcconfig` Files Are Your Friends

I love using `xcconfig` files. They are one of my favourite features of Xcode's build system. They allow me to define my build settings in a plain text file and customize how I want the values to resolve at build-time. To take full advantage of them I have removed all the values that are set in the project file and moved them into a series of xcconfig files. This removes a lot of the burden of managing build settings within Xcode and they exist only in these xcconfig files. One of the little-known features of the Build Settings view in Xcode is that if you select a row you can copy the name of the build setting and how it resolves for each configuration. Using this trick, it becomes very easy to migrate a target to using xcconfig values:

1. Open the project or workspace file in Xcode
2. Select the project file in the navigator panel
3. Select a target from the list that appears
4. Click on the tab labeled "Build Settings"
5. Select "Levels" and "All" options above the table that is now displayed
6. Create a new Xcode Configuation file and open that in another window or open another editor with a new document.
7. Select a row that has a green box around the bold lettering in the column that displays the current target's name. This should be next to the column that says "Resolved".
8. Copy the selected row, and paste it into the opened new configuration document that was created. The build setting should be pasted on one or more lines which values for each of the variation of that setting.
9. Going back to the Xcode window that displays the build settings, you can now press "delete" with that row selected to clear the values out of it.
10. Repeat this to migrate all your settings from the target in Xcode to the xcconfig file.
11. Once the migration is complete you will want to import all these settings so the target uses them again. To do this you will need to click on the project level settings (above the list of targets in Xcode)
12. Click on the "Info" tab at the top of the view now.
13. For each build configuration expand the collapsed list of targets, and assign the xcconfig file(s) you created to the targets that they correspond to.

There is a bit more to doing this process than described in the steps above, but it should give you a basic understanding of how to perform a migration to xcconfig files. For additional information on how to use xcconfig files you should check out the guide I wrote about them.

Project Level Settings

Additionally, moving all build settings that share the same values across all my targets onto the project level has been extremely useful for controlling how a build is performed. I do this to all the warning and error flags that can be set, within Xcode. This is one less thing I have to worry about and I can spend more time concentrating on driving down the number of warnings my code generates to keep it in good shape. Since any target in the project file will inherit the values set by the project's build settings, it takes some of the burden off of what you need to include in the xcconfig files.

My Project Still Has a Bad Merge

If you are diligent about following these practices you should almost never see a bad merge conflict happen. Sometimes you still will get them though. This will typically be when you are making non-trivial changes to the overall structure of the project file. To handle these situations my recommendation is to pick the version that is closest to what you want to have the project file represent, then perform any additional tweaks by hand. While this doesn't sound ideal there are few alternatives. Generally the structure of your project files should not be changing in any signficant manner often. If they are then I would suggest you look into alternative ways to manage building your targets (such as: make, cmake, autotools, ninja, etc). If you rely on Xcode's build system, then you should consider auto-generating your project files and not commit them into source control.
